Demonstrator prototypes of Computational Mechanisms of Interaction

Abstract

The present report documents the research activities undertaken in Task 3.3 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 presents various prototype systems based on the architecture and notation for constructing computational mechanisms of interaction. Each contribution focuses on one aspect of the work presented in previous deliverables. Finally, reference material and usage scenarios for each prototype are included.
© The COMIC Project, Esprit Basic Research Action 6225

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
Riso 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

Editor of this report:

Leandro Navarro
Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya
Gran Capità, s/n, Edifici D6-104
E-08071 Barcelona, Spain
Phone: +34-3-4016807
Fax: +34-3-4017055
Email: leandro@ac.upc.es

ISBN: 1-86220-003-3

Lancaster University, 1995
This report is available via anonymous FTP from ftp.comp.lancs.ac.uk.
Further information on COMIC is available from
http://www.comp.lancs.ac.uk/computing/research/cseg/comic/
Table of Contents

Introduction .................................................................................................................. 5

Part I. Mechanisms of Interaction in CSCW applications ......................... 13

Chapter 1. Coordination mechanisms: An approach to CSCW systems design ......................................................... 15

Chapter 2. Coordination Mechanisms in a multi-agent perspective ........... 73

Chapter 3. SOS as an implementation framework for the C-MOI notation ............................................................................. 119

Chapter 4. Let’s Talk About Bugs! (Towards Computer Support of the Coordination of Software Testing) ........................................... 143

Chapter 5. Object-Oriented Modeling of Coordination Mechanisms ....... 165

Chapter 6. BRaHS: A Computer Based Mechanism Supporting the Coordination of Bug-handling ......................................................... 183


Chapter 8. Demonstrating a cooperative system with an extended single user GUI ................................................................................. 255

Chapter 9. Design and Use of Mechanisms of Interaction ....................... 271

Part II. Demonstrator prototypes ................................................................. 305

Chapter 10. An executable specification of the agent architecture for Coordination Mechanisms ................................................................. 307

Chapter 11. The Aleph prototype: technical details .................................... 317
Introduction

The present report documents the research activities conducted within Task 3.3 of the COMIC project during the third year. It complements work presented in Deliverable 3.3.

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 3.3 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).

Activities in Strand 3 during the third year focused on three areas. First, further refinements and an exhaustive definition and formalisation of a complete notation for malleable and linkable mechanisms of interaction, as a refinement of work presented in Deliverable 3.3. Second, the development of computational demonstrators to validate the feasibility of the notation and architecture. Third, investigation of the applicability of this research to some environments such as the software development process, large learning networks, or the design of user interfaces.

Progress and results

This research produced the following results in the third year:

• The refinement of a complete notation for malleable and interoperable mechanisms of interaction.
• The definition of the Ariadne notation for the construction of malleable and linkable Computational Coordination Mechanisms, based on a multi-layer agent architecture.
• A prototype of a system based on work in Strand 3 and Strand 4. The Aleph architecture is a realisation of the SOS/SIS architecture developed in Strand 4, while the Aleph-Tcl language provides mechanisms for work articulation.
and a notation structured in consonance with the architecture and notation developed in Strand 3.

- A discussion on the applicability of the notation and the Aleph system to large scale distributed inter-networks exemplified in the I*EARN learning network.
- An analysis of coordination work in software testing in order to promote general requirements for computer support, and particularly, the identification of the need for computer support for coordinating the software testing process.
- An experimental investigation of how object-oriented analysis can contribute to modeling coordination mechanisms, based on a field study investigating the coordination of distributed software testing activities.
- A prototype of a system supporting software designers and testers in coordinating the process of reporting, diagnosing, and correcting software bugs. The system is called BRaHS (a Bug Reporting and Handling System).
- A concept demonstrator focusing on some of the architectural issues of providing computational coordination mechanism functionality supporting the coordination of distributed software testing through linking of active artifacts.
- A discussion of the shortcomings of using user interface objects as a basis for the MOI-tool design for demonstrating cooperative systems.

The structure of the deliverable

The structure of Deliverable 3.4 tries to separate the presentation of research results from the technical details of the demonstrator prototypes. Therefore Deliverable 3.4 is divided in two parts.

Part I: Mechanisms of Interaction in CSCW applications

Chapter 1: Coordination mechanisms: An approach to CSCW systems design

The paper outlines an approach to CSCW systems design based on the concept of ‘coordination mechanisms.’ The concept of coordination mechanisms has been developed as a generalization of phenomena described in empirical investigations of the use of artifactually imprinted protocols for the articulation of cooperative activities in different work domains. Based on the evidence of this corpus of empirical studies, the paper identifies a set of general requirements for computational coordination mechanisms and sketches the architecture of Ariadne, a CSCW environment for constructing such malleable and linkable computational coordination mechanisms.
Chapter 2: Coordination Mechanisms in a multi-agent perspective

The scope of the present chapter is to describe the main features of Ariadne, a notation for the construction of malleable and linkable C2Ms. The rationale behind Ariadne has been discussed in the previous chapter (§ 1.6). Here we illustrate how this rationale leads to a language formally defined both in its syntax and operational semantics. In doing this, we will follow the natural path connected with the structure of Ariadne layered in three levels (see figure 2.1.1, derived from chapter 1), by starting from the γ-level down to the α–level.

Chapter 3: SOS as an implementation framework for the C-MOI notation

The SOS is an architecture developed in the COMIC project to provide a rich computational environment for large and complex CSCW applications. This chapter describes how that architecture relates to the C-MOI notation. One realisation of that architecture is Aleph. The most important features of Aleph are the Resource Manager, the Aleph-Tcl notation and the mechanisms for linking environments (contexts and federation managers). Aleph-Tcl is a notation, an interpreted language to manipulate, configure, adapt and write new and existing Aleph components. It is compared with the C-MOI notation. Finally, the I*EARN network organisation is described in terms of the C-MOI notation, as an example of a real world environment where Aleph is being introduced.

Chapter 4: Let's Talk About Bugs! (Towards Computer Support of the Coordination of Software Testing)

Software testing is often a complex process potentially involving a large number of geographically distributed people with different perspectives and competencies. Software testers, software developers and project managers engage in discussions about the software errors found, they negotiate the relative importance of the bugs, they allocate responsibilities and resources, they coordinate who is doing what, etc. They talk about bugs. In order to coordinate and manage talking about bugs, a number of means for coordination are applied. The aim of this paper is to analyze coordination work in software testing in order to promote general requirements for computer support. We have studied the testing of more than 200,000 lines of code at Foss Electric, a Danish manufacturing company, and focused on two aspects: Firstly, the coordination activities related to the process of distributed registration, classification, diagnosis, correction, and verification of software errors, as well as the monitoring of the state-of-affairs of testing activities. Secondly, the mechanisms used to support the coordination. The analysis resulted in the identification of the need for computer support for coordinating this part of the software testing process, e.g., support of distributed classification, routing of information, and facilities providing an overview of state of affairs.
Chapter 5: Object-Oriented Modeling of Coordination Mechanisms

Computer support can enhance the effectiveness and efficiency of the coordination of distributed tasks. Analytical modeling of the work setting is an important prerequisite for designing computer-based coordination mechanisms. Object-oriented approaches to analysis, design and implementation have proven to be forceful means of modeling and analyzing complex systems. The aim of this paper is to experimentally investigate how object-oriented analysis can contribute to modeling coordination mechanisms. The experiment is based on the results from a field study investigating the coordination of distributed software testing activities. Previous identification and analysis of a set of linked coordination mechanisms is the starting point for the modeling experiment. These mechanisms consisted of paper and computer-based artifacts which were filled in and routed according to written procedures and culturally embodied conventions. The experiment resulted in an object model containing object-clusters, an object-class structure, and events and behavior diagrams. The experiment indicates that the modeling is useful for specifying the structural properties of coordination mechanisms as classes and objects. This facilitated decisions regarding how to link coordination mechanisms. The dynamic properties of coordination mechanisms reflecting interaction between actors are, however, not easily expressed in the models.

Chapter 6: BRaHS: A Computer Based Mechanism Supporting the Coordination of Bug-handling

This paper presents a prototype of a system supporting software designers and testers in coordinating the process of reporting, diagnosing, and correcting software bugs. The system is called BRaHS (a Bug Reporting and Handling System). The system can be regarded as a computer-based Coordination Mechanism that specifies the flow of the work and mediates relevant coordination information among the involved actors. Taking departure in empirically based requirements the functionality and structure of the system is described. It is then discussed and characterized by means of the central concepts in the conceptual framework of Mechanisms of Interaction: The three layered structure, the concepts of active artifacts, objects of articulation, the linking between mechanisms, and the overall requirements of local control, malleability, visibility, and flexibility, etc.

Chapter 7: Architectural Issues in Design of Computational Coordination Mechanisms for Software Testing

Previous chapters in this deliverable have reported from the analysis and design and implementation process aimed at demonstrating the concepts of computational coordination mechanisms (C2M) within the field of coordination of distributed software testing. This chapter documents the development of a concept demonstrator focusing on some of the architectural issues of providing computational coordination mechanism functionality supporting the coordination
Introduction

of distributed software testing through linking of active artifacts. We will in this chapter demonstrate the protocol of a set of C2M’s in an active artifact, the objects of articulation work in an example of notifications, malleability and linkability of the C2M’s. The application and its structure is, furthermore, related to the layered structure of the notation. Finally, we will reflect on the chosen architecture and on future developments by assessing the possibilities of using Apple Open Collaborative Environment and OpenDoc as a platform both for integrating computational coordination mechanisms, for providing malleability at the end-user level, and for providing flexible ways of linking C2M’s.

Chapter 8: Demonstrating a cooperative system with an extended single user GUI

During spring and summer 1994 a HyperCard based prototype software application was developed at University of Oulu for creating demonstrations of cooperative systems. This MOI-tool enables users to link user interface objects like fields, buttons, cards etc. between many workstations by slightly extending HyperCards facilities and its graphical user interface (GUI). The objective of the MOI-tool was to support demonstrating cooperative systems in a participatory design situation and the requirements to build it were derived from a small setting of case examples, whereas the ideas for implementation are principally loyal to user interface solutions of Hypercard. This chapter discusses of experimentally demonstrating a cooperative system: a bug handling mechanism, in order to discover differences between the MOI-tool design and the requirements of a possible cooperative system. The experiment results in one exemplary demonstration of a cooperative system with an explanation of breakdowns during this demonstration. These problems illustrate apparently the shortcomings of using user interface objects as a basis for the MOI-tool design for demonstrating cooperative systems. The paper ends up in a consideration of a possible future approach which could mean a change in concepts of tool design to another level of abstraction, in the level where we are discussing of roles, and actors, the objects of articulation work.

Chapter 9: Design and Use of Mechanisms of Interaction

This paper describes methods and tools for the development of CSCW applications. The development method is based on a framework for analysis of cooperative work that was used to analyse a number of case studies in Social Mechanisms of Interaction. On the basis of this framework and the notion of Mechanism of Interaction a software tool was designed and implemented that allows rapid construction of CSCW applications. A number of general issues concerning the notions of cooperation and articulation are discussed, and a tool is presented..
Part II: Demonstrator prototypes

The second part, Demonstrator prototypes of CMOI, supplements the first part with reference descriptions of each development and outlines scenarios of use of each prototype system:

Chapter 10: An executable specification of the agent architecture for Coordination Mechanisms

The paper presents the main features of the demonstrator under development, whose main goal 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 C2M according to the requirements stated in the field studies.

Chapter 11: The Aleph prototype: technical details

This chapter provides a brief technical overview of the Aleph prototype. Three aspects of the Aleph architecture are worth noticeable: (a) the Aleph Architecture, (b) the Aleph User Interface, and (c) the Aleph Language. The Aleph Architecture has been designed with large scale issues in mind. One concern has been to look at the relevance of a Resource Management as a function to support cooperative work in an organisational setting, and the concept of federation for the extension to an inter-organisational environment. The Aleph User Interface provides services for the interaction of people with several interfaces: (1) the Aleph-Tcl language interpreter, (2) the World-Wide-Web interface (3) the X-Windows interface, and (4) the UNIX shell interface. The Aleph Language is a notation based on the Tcl language to monitor, configure and re-specify the computational environment.

Future Challenges

This year, work has been concerned on the identification of general requirements for computational coordination mechanisms (C2M), the development of a computational notation with the required features, and the definition of computational architectures to construct CSCW environments with those characteristics. While this research was based on several field studies, future work must be addressed to complete the initial work on the construction and assessment of computational environments and demonstrators, in order to be in the position to assess the validity and usefulness of CSCW systems based on this approach. Only the development of well engineered software systems, would let us make further progress of the results achieved in this deliverable.

Experiences with the development and introduction of these CSCW systems on real world settings will provide very useful information to refine and assess the theoretical model, the computational architecture and the notation. This work
must be achieved by means of new field studies who will provide a measure of the usefulness of this research.

This would let us evaluate the generality of this approach as a general approach to CSCW system design, and in particular, on emerging environments where time, scale, or malleability are central issues.

In the area of software design, work should go in at least three directions: (1) to evaluate related products and concepts, (2) to tie our body of work to the ongoing efforts in conceptualizing relevant aspect of cooperative work, for example the concept of Coordination Mechanisms, and (3) to build experimental prototypes as a step forward in concretizing the ideas.
Part I: Mechanisms of Interaction in CSCW applications
Chapter 1

Coordination mechanisms:

An approach to CSCW systems design

Kjeld Schmidt
Risø National Laboratory

Carla Simone
University of Milano

The paper outlines an approach to CSCW systems design based on the concept of ‘coordination mechanisms.’ The concept of coordination mechanisms has been developed as a generalization of phenomena described in empirical investigations of the use of artifactually imprinted protocols for the articulation of cooperative activities in different work domains. Based on the evidence of this corpus of empirical studies, the paper identifies a set of general requirements for computational coordination mechanisms and sketches the architecture of Ariadne, a CSCW environment for constructing such malleable and linkable computational coordination mechanisms.

1. The issue of articulation work

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 were not 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 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 the sense that the field of work may have a multitude of possible states, the state of the field of work may be ambiguous, state changes may be interdependent in numerous ways, may occur intermittently and concurrently, and may be dynamic and unpredictable. Since members of such ensembles therefore are faced with complex interdependencies between individual activities, they cannot rely on accomplishing their individual and yet interdependent activities merely by changing the state of the field of work. They must articulate their distributed activities in other ways. With CSCW, these issues become crucial. In fact, CSCW can be conceived of as a field devoted to exploring how computer-based systems can enhance the ability of cooperating actors in articulating their activities (cf., e.g., Schmidt and Bannon, 1992; Fitzpatrick et al., 1995).

In order to be able to conceptualize and specify the support requirements of cooperative work, we make an analytical distinction between ‘cooperative work’ and ‘articulation work’. This distinction is fundamental to the approach presented here — and, in our opinion, to CSCW in general (Schmidt, 1994). Cooperative work is constituted by multiple actors who are interdependent in their work in the sense that they, elementally and fundamentally, interact through changing the state of the common field of work. When the activities are complexly interdependent, the distributed nature of the activities must be curbed through articulation work (Strauss et al., 1985; Gerson and Star, 1986; Strauss, 1988; Strauss, 1994) in the sense that the activities must be coordinated, scheduled, meshed, integrated etc. — in short: articulated.

**Proposition 1.** While cooperative work is constituted by multiple actors who are interdependent in their work in the sense that they interact through changing the state of the common field of work, articulation work is constituted by the need to curb the distributed nature of complexly interdependent activities.

The distinction between cooperative work and articulation work is recursive in the sense that an established arrangement of articulating a cooperative effort may itself be subjected to a cooperative effort which in turn also needs to be articulated, and so forth. To take a simple, and perhaps simplistic, example: at
some point during a design meeting one of the participants may interrupt the
design discourse in order to change the agenda for the meeting. Following that,
the participants discuss the proposal for some time, adopts it in an amended form,
and resume the design discourse where they broke off. In this case, the established
arrangement (the agenda) is treated as the field of work of another cooperative
effort, namely that of rearranging the agenda. This recursion is, in principle,
infinite. For instance, during the discussion about the proposed change to the
agenda, someone may raise the issue of floor control by, say, proposing that
nobody should be allowed to speak about the proposal more than once, which
may ignite a new round of exchanges at another level of recursion. While this
could go on forever, the infinite recursion is made finite and closed off, in order
to get the job done. The severe constraints under which work takes place in the real
world dictates that such recursions are terminated well before they become
frivolous.

\textit{Proposition 2.} Articulation work is a recursive phenomenon in that the
management of an established arrangement of articulating a cooperative
effort may itself be conducted as a cooperative effort which, in turn, may
also need to be articulated.

\section{The complexity of articulation work}

Cooperative work is inherently \textit{distributed}, not only in the usual sense that
activities are distributed in time and space, but also — and more importantly — in
the sense that actors are semi-autonomous in terms of strategies, heuristics,
perspectives, 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 uncertainties posed by the field of work and hence the variety of
heuristics involved, and so on. The more distributed the activities of a given
cooperative work arrangement, the more complex the articulation of the activities
of that arrangement is likely to be.

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 \textit{so} effectively and efficiently by our repertoire of
intuitive interactional modalities that the \textit{distributed} nature of cooperative work is
not routinely manifest. As demonstrated by the body of rich empirical studies of
cooperative work within CSCW, actors tacitly monitor each other; they perform
their activities in ways that support coworkers’ awareness and understanding of
their work; 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, and they mesh these interactional modalities dynamically and seamlessly (Harper et al., 1989; Heath and Luff, 1992; Harper and Hughes, 1993; Heath et al., 1995).

However, in face of the complex work settings that characterize modern industrial, service, and administrative organizations (hundreds or thousands of actors engaged in myriads of complexly interdependent activities), the task of articulating the interdependent and yet distributed activities is of an order of complexity where our everyday social and communication skills are far from sufficient.

In order to handle a high degree of complexity of articulation work, and handle it efficiently, the articulation of the distributed cooperative activities requires support by means of a special category of artifacts which, in the context of a set of conventions and procedures, stipulate and mediate articulation work and thereby are instrumental in reducing the complexity of articulation work and in alleviating the need for ad hoc communication.

Consider, for example, the case of the S4000 project. Foss Electric is a Danish manufacturing company producing advanced equipment for analytical measurement of quality parameters of agricultural products, e.g., the compositional quality of milk in terms of fat content and the count of protein, lactose, somatic cells, bacteria, etc.

At the time of the field study, the company was engaged in a large design project called S4000. The objective of the this project was to build a new instrument for analytical testing of raw milk. The S4000 project was the first project aiming at building an integrated instrument that would offer a range of functionalities that previously had been offered by a number of specialized instruments. In addition, as an innovation compared to previous models, the S4000 system would introduce measurements of new parameters in milk (e.g., urea and citric acid), and the measurement speed was to be radically improved. The instrument would consist of approximately 8,000 components grouped into a number of functional units, such as cabinet, pipette unit, conveyer, flow-system, and measurement unit. Finally, the S4000 was the first Foss instrument that incorporated an Intel-based 486 PC. The configuration and operation of the instrument was to be controlled via a Windows user interface. Eventually, the first version of the software consisted of approximately 200,000 lines of source code. Altogether more than 50 people were involved in the S4000 project, which lasted approximately 30 months (for version 1).

The design team was faced with quite a challenge:

1) The different subsystems, for example, the software control system and the mechanical and chemical processes in the flow and measurement system, were intricately interdependent and might interact in unforeseen ways.

---

1 The field research of the S4000 project was done by Henrik Borstrøm, Peter Carstensen, and Carsten Sørensen.
(2) The S4000 project introduced measurement of new parameters in raw milk for which new technologies had to be developed and mastered.

(3) The different subsystems were developed concurrently and the requirements to be satisfied by each subsystem would therefore change as other subsystems were developed.

(4) Production facilities at the manufacturing plant were constantly changing as the use of existing machines was optimized and new machines and processes were introduced. Hence, the repertoire of manufacturing processes that the production function could offer to designers and that designers thus had to take into account in their decisions was continually changing.

(5) Because of its technological heterogeneity, the S4000 project involved a number of specialities. The core design team consisted of designers from mechanical engineering, electronics, software, and chemistry. In addition, a handful of draught-persons and several persons from organizational entities such as production, model shop, marketing, quality assurance, quality control, service, and top management were involved to varying degrees at different stages in the course of the project.

All in all, the project was significantly more complex than previous projects at Foss.

In order to survive these challenges, the participants introduced a number of measures in order to reduce the complexity of managing the project:

Initially, all project participants from different technical departments were moved to the same office area in order to create a shared physical space by means of which participants could develop and maintain mutual awareness of the state of affairs. Furthermore, a sequence of meetings was scheduled at different intervals and, as the project took its course, a large number of ad-hoc meetings were arranged as well.

However, the amount of detailed information that had to be communicated, aligned, negotiated, etc., required more robust measures. A number of procedures and artifacts were introduced to keep track of the state of affairs, to manage relations and dependencies among involved actors, tasks, and resources: a bill of materials that identified actors responsible for parts in order to support the coordination of mechanical design, process planning, and production in the construction of prototypes (Sørensen, 1994a); a CEDAC board (Cause and Effect Diagram with the Addition of Cards) for coordinating the diagnosis of faults between mechanical design and process planning (Sørensen, 1994b); and a product classification scheme supporting the distributed classification and retrieval of CAD models (Sørensen, 1994c). Some of these procedures and artifacts were invented for this project, some were redesigns of existing artifacts, and others were merely adopted.

The most dramatic measures were taken with respect to the software design process. In the early phases of the software strand of the S4000 project, the participants felt that their overview of the state of the project was quite defective.
and that they needed much greater coordination. As one of the software designers put it:

‘It has really been problematic that we did not have any guidelines and descriptions for how to produce and integrate our things. The individual designers are used to work on their own and have all the required information in their heads, and to organize the work as they wish to […] When we started, we were only a few software designers. And suddenly — problems. And, oops, we were quite a few software designers and external consultants involved’.

At the height of the crisis the software design goals were almost abandoned. To overcome the crisis, the software designers developed a repertoire of procedures and artifacts to ensure the monitoring and control of the integration of software components and modules.

An important component in this repertoire, was the ‘software platform’ institution. Initially, the ‘software platform’ was just a point in time at which all software designers would stop coding in order to integrate their bits and pieces. For each platform integration period, one of the designers was appointed as ‘platform master’ which implied that he or she would be responsible for collecting information on changes made to the software and ensuring that the software was tested and corrected before it was released. Before the software was released as a ‘platform’ for further development, the project schedule was updated with revised plans and tasks for the next three to six weeks. The establishment of the software platform institution was considered absolutely necessary for the S4000 project.

Moreover, the software design team devised and introduced other procedures and artifacts: Most importantly, a ‘bug report form’, together with procedures for classifying, correcting, and reporting problems, was introduced in order to support the coordination of activities among software developers with respect to the detection and correction of software faults. More specifically, the purpose of introducing the bug form, was to ensure that bugs were properly registered and that corrected bugs were duly reported and to make the allocation of responsibilities among designers clear and visible to all designers. As a complementary measure, copies of bug forms that were being dealt with were collected in a publicly available repository in the form of a simple binder. (For further details, cf. Carstensen et al., 1995a).

The software designers experienced the hard way that it was practically impossible to handle the distributed testing and bug registration activities of some twenty testers and designers without, inter alia, such a bug report form and its associated procedures. By devising and introducing these constructs, they managed to alleviate the coordination crisis in the project.

The case of the S4000 project is particularly valuable because here we witness the emergence of novel coordination mechanisms in response to overwhelming problems encountered in coping with the complexities of articulating the distributed and interdependent activities of cooperative design under conditions that are typical for contemporary manufacturing. However, while daunting to the participants, the complexity of the S4000 project is not exceptional. Such
complexities are an everyday occurrence in modern industrial, service, and administrative settings.

Proposition 3. In cooperative work settings characterized by a high degree of complexity of articulation work, the articulation of the distributed activities requires artifacts which, in the context of a set of conventions and procedures, are instrumental in reducing the complexity of articulation work and in alleviating the need for *ad hoc* deliberation and negotiation.

Artifacts have been in use for coordination purposes in cooperative settings for centuries, of course — in the form of time tables, checklists, routing schemes, catalogues, classification systems in large repositories, and so on. Now, given the infinite versatility of computer systems, it is our contention that computer-based coordination mechanisms can provide a degree of visibility and flexibility to coordination mechanisms that was unthinkable with previous technologies, typically based on inscriptions on paper or cardboard. This opens up new prospects for 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 actor 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!¹

As a generalization, we call these artifacts and the concomitant procedures and conventions 'coordination mechanisms'.² In the following sections of this paper, we will expound this concept at length. However, it may be useful to summarize here the outcome of our subsequent discussion by anticipating the formal definition of a coordination mechanism:

Proposition 4. A coordination mechanism can be defined as a protocol, encompassing a set of explicit conventions and prescribed procedures and supported by a distinct 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 cooperative ensembles.

It follows from the fundamental distinction between ‘cooperative work’ and ‘articulation work’. (proposition 1) that, in order to be able to assist in the articulation of distributed activities in other ways than through changing the state

¹ For a related view of CSCW technology, cf. (De Michelis, 1994).
² In earlier papers we have used the term ‘mechanism of interaction’. The change of terminology does not imply any conceptual changes and is merely motivated by our experience that the term ‘mechanism of interaction’ can have unintended connotations.
of the field of work, the coordination mechanism must be *distinct from* the field of work.

*Proposition 5.* The coordination mechanism is *distinct from* 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 coordination mechanism and, conversely, changes to the state of the execution of the coordination mechanism are not automatically reflected in changes to the state of the field of work.

### 3. Coordination mechanisms: evidence and concept

The concept of coordination mechanisms has been developed as a generalization of phenomena described in different ways in numerous empirical investigations of the use of artifacts for coordination purposes 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);
- 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, 1994);
- schedules in hospital work (Zerubavel, 1979; Egger and Wagner, 1993);
- planning tools for manufacturing design (Bucciarelli, 1988; Sørensen, 1994a; Carstensen et al., 1995a);
- fault correction procedures in engineering and software design (Carstensen, 1994; Pycock, 1994; Pycock and Sharrock, 1994a; Sørensen, 1994b).

Consider, for example, the bug form as described in the study of the Foss Electric S4000 project:

The bug report form (see figure 1) is a two page form (both sides of one sheet of paper). It is filled-in, step-by-step, by the tester, the spec-team (three software designers responsible for diagnosing and classifying all reported bugs), and the designer responsible for correcting the bug. The bug report form is initially filled-in by anyone involved in testing the software, e.g., software designers, other designers, quality assurance staff, and marketing people. The originator of the bug form also provides a preliminary description and diagnosis of the problem. The ‘spec-team’ then specifies the module in which the presumably culpable module and the designer responsible, the platform integration period by which the bug should be corrected, and a classification of its perceived severity (as seen from
software reliability perspective). Finally, each designer was responsible for fixing the problems (handling ‘his’ or ‘her’ bugs forms) and reporting back to the Platform Master, i.e., the designer responsible for the next software module integration and verification period.

![Bug report form](image)

Figure 1. The bug report form (translated from Danish), with indications of which actor (role) is supposed to fill in which fields.

All bug forms were filed in a public repository (‘the binder’) which was organized according to the following categories: (1) non-corrected ‘catastrophes’, (2) non-corrected ‘semi-serious’ problems, (3) non-corrected ‘cosmetic’ problems, (4) postponed, (5) rejected, (6) corrected but not yet tested, and (7) corrected problems. The forms in the binder were successively re-classified and re-filed according to decisions made by the spec-team and messages concerning specific problems (bug forms) from the designers, results from the verification of the reported correction from the ‘platform integration period’, etc.

Finally, a list of not-yet-fixed problems (category 1, 2, and 3) was produced continually and accessed by the software designers. It gave them indications of the state of affairs in the software system as a whole and supported them in being aware of design activities with respect to modules for which they were not responsible but with which their own modules might interact.

Basically, five different roles were involved in coordinating the debugging activities in the S4000 project: the testers testing and reporting the software, the ‘spec-team’ diagnosing the reported bugs, the software designers correcting the
diagnosed bugs, the ‘platform master’ verifying the corrected bugs, and the designer maintaining the bug form repository in order to keep track of the distributed debugging activities (see figure 2).

Figure 2. A schematic illustration of the roles and information flows in software testing in the S4000 project. The flows in the diagram indicates the intended flow of the bug handling protocol. The protocol follows seven major steps: (1) a tester sends a form, describing a bug, to the ‘spec-team.’ (2) the ‘spec-team’ diagnoses the bug and sends the annotated form to a software designer; (3) a copy is send to the bug form repository manager; if a bug is rejected, the original is sent to the bug form repository manager; (4) the software designer enters information on corrections and sends the form to the bug form repository manager; (5) the bug form repository manager sends a stack of forms of bugs to be verified to the ‘platform master’; (6) forms of bugs that cannot be verified are sent to the ‘spec-team’; (7) verified forms are sent to the bug form repository manager.

— One designer aptly described this protocol as ‘a state-transition diagram with more than fifteen states’ (Carstensen et al., 1995b).

As illustrated by the case of the bug form, a coordination mechanism can be conceived of as constituted by two distinct devices: on one hand a protocol in the form of a set of procedures and conventions which, to competent members, stipulate the responsibilities of different roles, the classifications of bugs, the intricate flow of forms, acknowledgments, reports on corrected bugs, and so forth, and on the other hand the bug form as an artifact, i.e., as a distinct and durable symbolic construct. Moreover, the bug form and the protocol are intimately coupled in the sense that the components of the artifact as a symbolic construct (i.e., the different fields) stand proxy for key components of the protocol. In other words, the protocol can be said to be ‘imprinted’ upon the artifact whereas the artifact serves as ‘a window’ to the protocol.

Proposition 6. A coordination mechanism is an integral construction consisting of a protocol (an integrated set of procedures and conventions stipulating the articulation of distributed interdependent activities) on the one hand and on the other hand an artifact (a permanent symbolic construct) in which key components of the protocol are imprinted.
Let us analyze these constituent parts in turn.

3.1. Coordination mechanisms: the protocol

While the notion of protocols that stipulate the articulation of cooperative work is crucial to the concept of coordination mechanisms, it is also contested. In a large body of sociological literature, the notion of pre-defined organizational constructs (formal structures, procedures, methods, plans) as determinants of action has been subjected to critical examination. Study after study has demonstrated, unambiguously and beyond any doubt, that the status of these formal organizational constructions in the actual course of work is problematic in the sense that these constructions are impoverished idealizations when taken as representations of actually unfolding activities. In the words of Philip Selznick’s classic summary of this line of sociological investigation:

‘The formal administrative design can never adequately or fully reflect the concrete organization to which it refers, for the obvious reasons that no abstract plan or pattern can — or may, if it is to be useful — exhaustively describe an empirical totality. At the same time, that which is not included in the abstract design (as reflected, for example, in a staff-and-line organization chart) is vitally relevant to the maintenance and development of the formal system itself.’ (Selznick, 1948, p. 25)

This conception of the status of ‘formal constructions’ has been highly influential in that it has, as observed by Egon Bittner in a now classic paper, ‘furnished the necessary theoretical argument for an entire field of sociological investigations by directing attention to a sphere of adaptive and cooperative manipulations, and to the tensions typically found in it.’ (Bittner, 1965, p. 240)

Now, how do sociologists go about distinguishing the facts of formal organization from the facts of informal organization? The methodological rule for making this distinction can, in Bittner’s words, be stated this way:

‘In certain presumptively identified fields of action, the observed stable pattern of conduct and relations can be accounted for by invoking some programmatic constructions that define them prospectively. Insofar as the observed stable patterns match the dispositions contained in the program they are instances of formal organizational structure. Whereas, if it can be shown that the program did not provide for the occurrence of some other observed patterns which seem to have grown spontaneously, these latter belong to the domain of the informal structures.’ (Bittner, 1965, p. 240)

The issue for Bittner — and for us — is: what is the status of these formal organizational constructions? The problem with the received tradition of critical studies of formal organizational constructions, however, is the almost ceremonial status it implicitly ascribes to these formal constructions and the ensuing dichotomy of the ‘formal’ and the ‘informal’, the notional and the corporeal. The argument implies that members of the organizational settings in question are somehow supposed to take formal constructions literally — as if constructions such as procedural formulations are supposed to be exhaustive specifications of how the work gets done.

In addressing this problem, Bittner makes some very cogent observations:
‘While Selznick quite clearly assigns the formal schemes to the domain of sociological data, he does not explore the full range of consequences out of this decision. By retaining Weber’s conception of them as normative idealizations, Selznick avoids having to consider what the constructions of rational conduct mean to, and how they are used by, persons who have to live with them from day to day. It could be, however, that the rational schemes appear as unrealistic normative idealizations only when one considers them literally, i.e., without considering some tacit background assumptions that bureaucrats take for granted.’ (Bittner, 1965, p. 242 - emphasis added)

Bittner’s methodological recommendation is quite pertinent to the issue of analyzing and designing coordination mechanisms. In order to be able to contribute constructively to the design of computational coordination mechanisms, we need to understand not only ‘the tacit background assumptions’ that members take for granted and without which any formal construction would be merely a rhetorical statement but also ‘what the constructions of rational conduct mean to, and how they are used by, persons who have to live with them from day to day’.

In the course of the following exposition and discussion, it is important to notice that we are not trying to solve or address the general problems of general sociological theory. We are not investigating human action in general, merely the means of articulating distributed and interdependent activities in work settings, that is, under conditions of severe constraints. Nor are we investigating not the nature of tacit and implicit plans, rules, routines, habits, and so on in human social conduct in general but, more specifically and modestly, the role of artifactually imprinted protocols deliberately designed and used to support the articulation of cooperative work.

During the last 15 years or so, our understanding of how procedures and artifactually imprinted protocols are used by actors in everyday work activities has been greatly enriched by a number of outstanding studies (e.g., Wynn, 1979; Suchman, 1983; Suchman and Wynn, 1984; Bucciarelli, 1988; Wynn, 1991). The general conclusion of these studies is that such procedures and artifacts serve as ‘maps’ (Suchman, 1987, p. 188 f.; Bucciarelli, 1988, p. 114).

Consider, for instance, Suchman’s study of the accounting office (Suchman, 1983): This office was responsible for the orderly payment of money due to outside organizations supplying goods and services to the organizational units in its charge. Orderly payment was documented through record-keeping, and accuracy was monitored by the auditing of invoices against records of requisition and receipt.

According to the standard procedure, items on a given purchase order could be received and billed in separate installments over an extended period of time. Again, if all went smoothly, the items marked off on the receiving report from Shipping/Receiving would correspond to those on the invoice from the vendor. The purchase order, receiver, and invoice would be matched and audited. The payment for the items received would be recorded by margin notes on the purchase order, which would then be returned to the temporary file to wait for the
next shipment and billing. Only after all bills had been received and paid was the completed purchase order filed permanently in the paid file.

In the case presented and analyzed by Suchman, however, the record of what had happened was incomplete: The original purchase order was missing. A completed receiving document was found with eight items listed on it, all of which had been marked as received. But the two invoices found in the paid file showed only two items as paid; there was no invoice or record of payment for the other items, yet the vendor reported that the transaction would be completed with payment of the past due invoice for only two of those items that seemingly had not yet been paid. The study then shows how the two actors, the accounting clerk and the auditing clerk, step by step solved the ‘mystery’: Of the invoice for one of the items, only page two is on file; page one is missing. It thus transpired that four other items were invoiced with this item and had already been paid.

This case shows convincingly that orderly records are not necessarily the result of some prescribed sequence of steps and that may involve the practice of completing a record or pieces of it after the fact of actions taken: ‘once the legitimate history of the past due invoice is established, payment is made by acting as though the record were complete and then filling in the documentation where necessary. The practice of completing a record or pieces of it after the fact of actions taken is central to the work of record-keeping’ (Suchman, 1983, p. 326). Thus, precisely because it is a case of recovery from error, the case gives a vivid impression of the massive heuristic use of standard procedures even in a seemingly abnormal situation. The two actors are able to solve the abnormal problem because of their ‘knowledge of the accounts payable procedure’ (Suchman, 1983, p. 322). Standard procedures can thus be said to have a heuristic function in the sense that they ‘are formulated in the interest of what things should come to, and not necessarily how they should arrive there’ (Suchman, 1983, p. 327).

Taking this interpretation further, Suchman posits that a standard procedure serves as an extraneous and subservient referent for situated action:

‘It is the assembly of orderly records out of the practical contingencies of actual cases that produces evidence of action in accordance with routine procedure. This is not to say that workers ‘fake’ the appearance of orderliness in the records. Rather, it is the orderliness that they construct in the record that constitutes accountability to the office procedures.’ (Suchman, 1983, p. 327)

In her seminal book on Plans and Situated Action (1987) this interpretation is generalized:

‘plans are resources for situated action, but do not in any strong sense determine its course. While plans presuppose the embodied practices and changing circumstances of situated action, the efficiency of plans as representations comes precisely from the fact that they do not represent those practices and circumstances in all of their concrete detail’ (Suchman, 1987, p. 52).

Suchman’s thesis that ‘plans are resources for situated action’ is of fundamental importance to CSCW systems design and has served us as a guiding
principle in the development of the concept of coordination mechanisms. But it also leaves a number of unsettling questions unanswered: What is it that makes plans such as production schedules, office procedures, classification schemes, etc. useful in the first place? What makes them ‘resources’? Furthermore, is it merely the fact that plans are underspecified in comparison with the rich multiplicity of actual action that makes them ‘resources’? Is that really all there is to it? What, then, makes one procedure or form or schedule more useful than another for a certain purpose in a specific setting?

Later in the book, Suchman returns to these issues and suggests a rather apt metaphor for the role of artifactually imprinted protocols, namely that of a ‘map’:

‘Just as it would seem absurd to claim that a map in some strong sense controlled the traveler’s movements through the world, it is wrong to imagine plans as controlling actions. On the other hand, the question of how a map is produced for specific purposes, how in any actual instance it is interpreted vis-à-vis the world, and how its use is a resource for traversing the world, is a reasonable and productive one.’ (Suchman, 1987, pp. 188 f.)

While the same irksome questions arise here as well, the ‘map’ analogy is a fitting condensation of the role of artifactually imprinted protocols that have been described in a number of studies. In Suchman’s study of the accounting office, for example, the standard operating procedures were found to be ‘formulated in the interest of what things should come to, and not necessarily how they should arrive there’. They were used as a general reference for orientation purposes, not as a prescribed sequence of actions to be taken.

However, other studies lead to quite different conclusions as to how artifactually imprinted protocols are used by actors in everyday work activities.

Checklists. Firstly, consider the relatively simple case of the ‘normal checklist’. The checklist is an artifactually imprinted protocol that has been deliberately and carefully designed to reduce local control in safety-critical environments. More specifically, a checklist is used to organize tasks whenever it is essential that a set of actions all be performed, typically where it is essential that the actions of the performance also be taken in a particular order, in order to ensure a high level of operational safety. For example, the normal aircraft flight-deck checklist indicates a set of different tasks the pilot must perform or verify during all flight segments in order to configure the aircraft and prepare the flight crew for certain ‘macro-tasks’ such as ENGINE START, TAXI, TAKEOFF, APPROACH, LANDING, etc. For each one of these macro-tasks there are several ‘items’ to be accomplished and verified by the crew (Degani and Wiener, 1990).

In his analysis of the checklist, Don Norman observes ‘The fact that the preparation of the list is done prior to the action has an important impact upon performance because it allows the cognitive effort to be distributed across time and people’ (1991, p. 21). This preparatory task — which Hutchins and Norman, in another study, aptly call ‘precomputation’ — can be done when more

---

1 Is it indeed “precisely […] the fact that they do not represent those practices and circumstances in all of their concrete detail” that makes plans efficient and effective? Does that mean that the less specific the better? Suchman probably does not intend to imply that.
convenient, e.g., when there is no time pressure and no safety and security risk, and by another actor, e.g., by a specialist. ‘In fact,’ Norman observes, ‘precomputation can take place years before the actual event and one precomputation can serve many applications’ (1991, p. 21). The concept of a precomputation of essential aspects of a task is crucial to understanding the role of artifactually imprinted protocols: The flight-deck checklist, for instance, provides a precomputed selection of safety-critical tasks, which all need to be performed at the particular flight segment as well as a precomputed sequence for their execution.

Now, the protocol of the flight-deck checklist does not stipulate the articulation of cooperative activities but the activities themselves. For a case of the use of artifactually imprinted protocols that stipulate the articulation of cooperative activities, consider the kanban system.

The kanban system. In 1990, Bjarne Kaavé conducted a study of cooperative production control in a manufacturing company we can call Repro Equipment. The company manufactured specialized optical appliances, and it covered about 50% of the world market for this category of equipment. At the time of the study company produced about 6,000 units a year in fifteen models, each in seven different variants.

A manufacturing operation, like the one at Repro Equipment, involves multitude discrete parts and processes that are complexly interdependent: Each product consists of a many component parts, in some cases tens or hundreds of thousands of components, and their production may require a number of different processes in a specific sequence. Different processes, such as cutting, bending, welding etc., typically require specialized tools and skills which are distributed at different workstations and require hugely different set-up times. This is compounded by the fact that, at any given time, a large number of products and their components coexist in the production process at different stages of completion which means that different parts for the same or for different products compete for the same workstations. Thus, in the words of Harrington (1984), manufacturing can be conceived of as ‘an indivisible, monolithic activity, incredibly diverse and complex in its fine detail. The many parts are inextricably interdependent and interconnected.’ Accordingly, for a manufacturing enterprise to be able to adapt to changing conditions, the entire enterprise must react ‘simultaneously and cooperatively’ (Harrington, 1979, p. 35).

To deal with this complexity, Repro Equipment used a kanban system to coordinate processes in the manufacture of cabinets. Kanban is a Japanese word meaning ‘card’ or more literally ‘visible record’ (Schonberger, 1982, p. 219) and it is now in widespread use in manufacturing to denote a just-in-time production control system where a set of cards acts as the carrier of information about the state of affairs as well as a production order conveying an instruction to initiate

---

1 This analysis is based on Bjarne Kaavé’s findings as reported in his thesis (Kaavé, 1990) as well as in several joint analysis sessions with one of the present authors.
certain activities. The basic idea is that loosely coupled but interdependent production processes can be coordinated by exchanging cards between processes. When a new batch of parts or sub-assemblies has been produced and the batch is to be transported ‘down-stream’ from the present work station to the station where it is to be used, for instance, as components for various sub-assemblies, a specific card is attached to the container used for the transportation. When the operator at the work station down-stream has processed this batch of parts, the accompanying card is sent back to the operator who produces these parts. To the operator, receiving the card means that he or she has now been issued a production order.

The basic set of rules of a kanban protocol are as follows (Schonberger, 1982, p. 224):

1. No part may be made unless there is a kanban authorizing it.
2. There is precisely one card for each container.
3. The number of containers per part number in the system is carefully calculated.
4. Only standard containers may be used.
5. Containers are always filled with the prescribed quantity — no more, no less.

Setting up a kanban system requires a careful configuration of the number of containers per part number and the quantity per container. This configuration, in effect, amounts to a precomputation of tasks in terms of batch size per part number, task allocation in terms of work stations for different part numbers, and task sequences.

However, the kanban system is not adequate for coordinating manufacturing operations faced with severe demands on flexibility of volume: a kanban system can only handle small deviations in the demand for the end product (Schonberger, 1982, p. 227; Monden, 1983). Accordingly, since Repro Equipment was faced with extreme differences and fluctuations in demand for different models and variants, operators recurrently experienced that the configuration of the kanban system (the number of containers per part number and the quantity per container) was inadequate. For instance, in a situation where a particular part number that was only used for a special product variant had all been used, the protocol would automatically generate a production order for this part number, irrespective of the fact that the part number in question probably would not been needed in months and would thereby absorb production facilities that would be needed for other, more pressing orders.

In such situations, where the kanban system is ‘beyond its bounds’, operators at Repro Equipment would tamper with the kanban protocol. For example, having heard of a new rush order from the girl in the order office, the fork lift operator might put the card for a rarely used part for another model in his back pocket or leave it on the fork-lift truck for a while. Similarly, in order to rush an order, operators would occasionally order a new batch of parts for this order before the
container had actually been emptied and the card had been released, or they would deviate from batch sizes as specified on the card, etc.

It is crucial to notice that in stead of abandoning the kanban system altogether, or at least temporarily, the operators changed the configuration of the system. That is, when an operator pocketed a kanban, he or she was modifying the protocol, not switching it off, and when the card was put back in circulation (or released belatedly), the default configuration was in force again. The reason for this is that the kanban system incorporates (implicitly, in the configuration of the system) a precomputed model of crucial interdependencies of the manufacturing process (routing scheme, set-up-times etc.). Thus, in spite of the fact that the kanban system at Repro was often used in situations where it was ‘beyond its bounds’, it was not discarded but merely modified locally and temporarily according to the requirements of the situation.

In order to be usable in a setting like Repro Equipment, the kanban system had to be managed (monitored, adapted, modified) continually. This was facilitated by the formation of a network of clerks, planners, operators, fork-lift drivers, and foremen in various functions such as purchasing, sales, production, shipping etc. who kept each other informed about the state of affairs so as to control the flow of parts. A member of this network would for example explore the state of affairs ‘up-stream’ so as to be able to anticipate contingencies and, in case of disturbances that might have repercussions ‘down-stream,’ issue warnings. That is, the indirect, dumb, and formal kanban mechanism was subsumed under a very direct, intelligent, and informal cooperative arrangement. The cooperative ensemble ‘appropriated’ the kanban system in order to increase its flexibility. They took over control of the system and controlled production far more closely and effectively than warranted by the design of the kanban system.¹

Coming back to the issue of the status of formal organizational constructions in cooperative work, the kanban system illuminates several important points.

Suchman’s contention that the function of abstract representations such as plans ‘is not to serve as specifications for the local interactions, but rather to orient or position us in a way that will allow us, through local interactions, to exploit some contingencies of our environment, and to avoid others’ (Suchman, 1987, p. 188) is not correct as far as the kanban system is concerned. When an operator receives a card, he or she will produce the batch as specified by the card, in accordance with the general rules of the protocol, without actively searching for reasons not to do so and without deliberating or negotiating whether to do so or not.

¹ Notice that the kanban system might not work like this if it had been designed otherwise, for instance with sensors at the bottom of each container in order to detect and notify automatically when a new production order should be issued. This might make it difficult for operators to change the configuration contingently.
In their individual activities, actors rely on the *kanban* system to issue valid and sensible production orders, unless they have strong reasons to believe that its unmitigated execution in the particular situation at hand will have undesirable results. Even then, they do not discard the system but alter its behavior by reconfiguring it, after which the system is allowed to ‘switch back’ to the default configuration. That is, in the case of the *kanban* system:

- actors coordinate their distributed activities by executing the *kanban* protocol — unless they have strong reasons to act otherwise;
- when actors have reasons to doubt the rationality of executing a production order issued by the system, they temporarily reconfigure the system, i.e., respecify the protocol, by withholding cards or introducing false cards;
- by reconfiguring the system, actors do not discard the system but alter its behavior temporarily, upon which the system is allowed to ‘switch back’ to its default configuration.

The *kanban* system thus determines action a far stronger sense than the map of a traveler determines the traveler’s movements (Suchman, 1987, p. 188 f.; Bucciarelli, 1988, p. 114). In the *kanban* case the protocol conveys a specific stipulation in the form of a production order to the particular actor instructing the actor, under the conditions of social accountability, to take the particular actions specified by the card according to the general rules of interpretation laid down in the protocol. It is thus more like a *script* than a *map*. In fact, the *kanban* system works well even though it does not provide a ‘map’ in the form of an overview of interdependencies among processes.

The point is that the *kanban* protocol under normal conditions of operation relieves actors of the otherwise forbidding task of computing myriad — partly interdependent, partly competing — production orders and negotiating their priority. They can, for all practical purposes, rely on the precomputed protocol to issue valid production orders; they take it for granted. Thus, for an actor in Repro Equipment to question the rationality of the protocol at every step in every situation would be an utter waste of effort, and it does not happen.

As a generalization, we find that a protocol stipulates the articulation of distributed activities by conveying affordances and constraints to the individual actor which the actor, as a competent member of the particular ensemble, can apply without further contemplation and deliberation unless he or she, again as a competent member, has accountable reasons not to do so. That is, actors deviate from the stipulations of the protocol if and when they have compelling reasons to do so, and only then.¹

**Proposition 7.** A coordination mechanism is a resource for situated action in the sense that it reduces the complexity of articulating cooperative work by

¹ “Even the simple checklist reduces the semantic distance for its users. Lacking the checklist, the novice must discover the steps that need to be done and an order in which they can be applied. With the checklist, the task is transformed: reading and following instructions take the place of procedural reasoning.” (Norman and Hutchins, 1988, p. 15)
providing a precomputation of essential interdependencies among cooperative activities which the individual actor, faced with intricately interlaced constraints and possibilities, for all practical purposes can rely on to reduce the space of possibilities by identifying a valid and yet limited set of options for coordinative action.

As demonstrated by the conflicting findings from different cases, artifactually imprinted protocols, such as plans, conventions, procedures, and so forth, play different roles in cooperative work. They may, on one hand, play the ‘weak’ role of the ‘map’ of the traveler by providing a codified set of functional requirements which provides a general heuristic framework for distributed decision making. On the other hand, they may play the ‘strong’ role of a ‘script’ that offers a precomputation of interdependencies among activities (options, sequential constraints, temporal constraints, etc.) which, for each step, provides instructions to actors of possible or required next steps. Which role is appropriate naturally depends on the extent to which it is possible to identify, analyze, and model interdependencies in advance.

Moreover, the role of a particular protocol may vary according to the situation. Thus, in a situation where a standard operating procedure does not apply, the procedure may merely serve in its weak default capacity as a vehicle of conveying heuristics (as, for instance, in the accounting office). In other cases, however, such as the kanban case, the role of the protocol does not vary in the face of contingencies; rather, because of the complexity of the interdependencies of discrete parts production, the kanban protocol was not discarded, suspended, nor ‘weakened’ but temporarily respecified (reconfigured) by operators to accommodate the passing disturbance.

**Proposition 8.** The role of artifactually embodied protocols in stipulating the articulation of cooperative work varies from case to case and from situation to situation, according to the fitness and expressive power of the precomputed interdependencies as represented by the protocol, from the weak stipulations exemplified by ‘a map’ to the strong stipulations, exemplified by ‘a script’, that offers a precomputation of interdependencies among activities which determines action in the sense that it, for each step, provides instructions to actors of possible or required next steps.

Weak or strong, protocols are, as pointed out by Suchman, inexorably characterized by ‘the inherent and necessary under-specification of procedures with respect to the circumstances of particular cases’ (Suchman, 1982, p. 411). Furthermore, Suchman observes, ‘the vagueness of plans is not a fault, but is ideally suited to the fact that the detail of intent and action must be contingent on the circumstantial and interactional particulars of actual situations’ (Suchman, 1987, pp. 185 f.). However, the degree of vagueness of specific plans is itself contingent:
‘While plans can be elaborated indefinitely, they elaborate actions just to the level that elaboration is useful; they are vague with respect to the details of action precisely at the level at which it makes sense to forego abstract representation, and rely on the availability of a particular embodied response.’ (Suchman, 1987, p. 188)

Thus, it is not only that a protocol, as a linguistic construction (Suchman, 1987, p. 186), is inherently vague compared to the rich details of the actually unfolding activities of the cooperative work arrangement in which it is applied, nor is it only that a protocol is inherently decontextualized, but a protocol is deliberately under-specified with respect to (a) factors that are immaterial for the purpose of the given protocol or (b) factors that can more efficiently and effectively be left unspecified, typically until a later stage. The protocol must, to use the apt phrase of Bowker and Star, be defined at ‘an appropriate level of ambiguity’ (Bowker and Star, 1991, p. 77).

Proposition 9. As a preconceived plan for the articulation of the distributed activities of a specific cooperative work arrangement, a coordination mechanism is inexorably under-specified in the sense that the nominal preconception cannot encompass and denote the infinite multiplicity of actual circumstances and occurrences unfolding during its situated enactment.

Thus, weak or strong, the execution of a protocol, involves an unavoidable element of situated interpretation and improvisation.

On the other hand, weak or strong, the protocol will, inevitably, be faced with situations where it is beyond its bounds, its inherent vagueness and appropriate ambiguity notwithstanding. This is eloquently illustrated by the case of the kanban system. Similarly, the software designers intermittently experienced situations where the bug form protocol they had devised and adopted did not provide adequate stipulations and the execution of the bug form protocol was thus negotiated and modified as a new designer became involved, as the classification of a particular bug was found to be erroneous, as a designer rejected the responsibility ascribed to him by the spec-team, and so forth.

Proposition 10. Whether weak or strong, a coordination mechanisms will, inevitably, be faced with situations where it is beyond its bounds and where actors therefore must modify the execution of the protocol.

Let us now turn to the role of the artifact in coordination mechanisms.

---

1 The written medium encourages the decontextualization or generalization of stipulations of orderly cooperative work. In their very nature, written stipulations have been abstracted from particular situations in order to be addressed to the target audience in general, rather than delivered face-to-face to a specific group of people at a particular time and place (Goody, 1986, pp. 12 f.).
3.2. Coordination mechanisms: the artifact

The role of the artifact in coordination mechanisms is, fundamentally, to give permanence to the protocol for which it stands proxy in the sense that it conveys the general and situation-independent stipulations of the protocol. So far, the artifact is merely a written record of the precomputations of the protocol.

While written language, as observed by Jack Goody, ‘is partly cut off from the context that face-to-face communication gives to speech, a context that uses multiple channels, not only the purely linguistic one, and which is therefore more contextualized, less abstract, less formal, in content as in form.’ (Goody, 1987, p. 287), written records (log books, recordings, minutes, memos etc.) provide persistence to decisions and commitments made in the course of articulation work: ‘The written language [reaches] back in time’ (Goody, 1987, p 280). Written records are, in principle, accessible to any member of the ensemble, whatever its size and distribution in time and space. In the words of Stinchcombe, ‘Written systems can provide a larger number of people with the same information at one time’ and written messages are ‘portable, allowing interaction without spatial constraints.’ On the other hand, written systems ‘are much less dependent on physical arrangements’ and ‘less time-dependent than oral systems.’ (Stinchcombe, 1974, pp. 50 f.). Written artifacts can at any time be mobilized as a referential for clarifying ambiguities and settling disputes: ‘while interpretations vary, the word itself remains as it always was. (Though every reading is different, it is a misleading exaggeration of the literary critic to say that the text exists only in communication.)’ (Goody, 1986, p. 6). They are, for all practical purposes, unceasingly publicly accessible.

Proposition 11. The role of the artifact in a coordination mechanism is fundamentally to give permanence to the protocol so that its stipulations are unceasingly publicly accessible.

Consider, for example, a standard operating procedure or a checklist. The state of the artifact is completely static irrespective of the state of the execution of the protocol it prescribes. Even when the artifactually imprinted protocol is used as a script (actors are following the instructions of the procedure or the items of the checklist step by step), it is entirely up to the actor to produce and maintain the required dynamic representation of the state of the protocol with respect to the unfolding cooperative activities.

In the case of the bug report, however, the state of the artifact changes according to the changing state of the protocol. Firstly, the form is transferred from one actor to another and this change of location of the artifact in itself conveys, to the recipient, the stipulations of the protocol in a specified form, in the sense that the change of location transfers to the particular actor the specific responsibility of taking such actions on this particular bug that are appropriate according to the agreed-to protocol and other taken-for-granted conventions. Secondly, at each step in the execution of the protocol, the form is annotated and
the thereby updated form retains and conveys this change to the state of the protocol to the other actors — the state of each reported bug is thus reflected in the successive inscriptions on the form made by different actors. That is, a change to the state of the protocol induced by one actor (a tester reporting a bug, for example) is conveyed to other actors by means of a visible and durable change to the artifact. Furthermore, this change is propagated within the ensemble according to the stipulations of the protocol, and the state of the total population of reported bugs is publicly visible in the public repository of bug forms (‘the binder’).

Similarly, in the case of the kanban mechanism, the artifact mediates articulation work in the sense that the change of location of a card, that is, the fact that it is transferred ‘up-stream’ from one actor to another, is equivalent to the arrival of a production order at that work station. However, as opposed to the bug report, the inscription on the kanban card is not changed and the state of the kanban protocol is thus not reflected in any particular card. Hence, state changes to the protocol under execution can not be inferred from the inscription on the cards, only from their location.

In these cases, the artifact not only stipulates articulation work (like a standard operating procedure) but mediates articulation work as well in the sense that the artifact acts as an intermediary between actors that conveys information between them about state changes to the protocol under execution.1 By serving the dual function of stipulating and mediating articulation work, the artifact is instrumental in reducing the complexity of articulating a vast number of interdependent and yet distributed and perhaps concurrently performed activities.

**Proposition 12.** The artifact of a coordination mechanism may, in some form and at a particular level of granularity, dynamically represent the state of the execution of the protocol and may thereby serve as an intermediary between actors that mediates information between them about state changes to the protocol as it is being executed.

Due to the artifact mediating the changing state of the protocol between actors, the coordination mechanism not only conveys the general stipulations of the protocol but specifies the stipulations in the sense that the individual actor is instructed that it is he or she that has to take this or that specific action at this particular point in time. In other words, by representing and conveying the

---

1 Edwin Hutchins has a related but not identical analysis of artifactually imprinted protocols. In his discussion, Hutchins suggests the term ‘mediating structures’ for artifacts that are not part of the field of work (as tools are) and yet are instrumental in reducing the complexity of work by providing some kinds of constraints to the conduct of the actor (Hutchins, 1986, p. 47). For Hutchins, the artifact or structure serves as an intermediary between an actor planning or defining the protocol for an activity and the actor performing the activity, whereas we, for our purposes, reserve the term ‘mediate’ to denote an artifact serving as an intermediary of horizontal propagation of state changes to the protocol, i.e., within the cooperative work arrangement at hand. In other words, coordination mechanisms can be conceived of as a special case of ‘mediating structures’, namely artifactually embodied ‘mediating structures’ that are used to constrain the articulation of distributed activities in cooperative work settings.
changing state of the protocol, the artifact also mediates the transition from nominal to actual in the enactment of the protocol.

**Proposition 13.** By mediating the changing state of the protocol, the artifact of a coordination mechanism specifies the general stipulations of the protocol.

Due to its mediating role with respect to the state of the execution of the protocol, the artifact may support the development and maintenance of mutual awareness among the actors within the cooperating ensemble. By reflecting the state of the execution of the protocol, the artifact may convey information about occurrences within the ensemble from which actors can make inferences about likely or possible problems and develop an overview of the state of the protocol in its totality. This potential is clearly illustrated by the case of software testing, especially due to the successive inscriptions on the bug forms and the ordered assembly of (copies of) bug forms in ‘the binder’. The *kanban* system, on the other hand, does not provide a facility for obtaining such an overview. In fact, the information conveyed by the transfer of cards up-stream is drastically filtered and distorted by the successive translations from card to card. The only interface to the state of the protocol across the total population of cards in circulation is the (ever changing) location of the myriad cards in the distributed manufacturing system. That is, the *kanban* system does not provide facilities allowing actors to develop and maintain a mutual awareness so as to, for instance, anticipate disturbances and obtain an overview of the situation within the cooperative ensemble at large; they are, so to speak, enveloped by an overwhelming and inscrutable quasi-automatic coordination mechanism. — In fact, in the *kanban* system, changes to the state of the artifact are strongly coupled to state changes in the field of work. Information only propagates ‘up-stream’ as parts are used down-stream: the speed and pattern of propagation of information are thus restricted by the rate and pattern of changes to the field of work at large. The *kanban* system does allow operators to control of the execution of the protocol, however, since that control is ultimately in the hands of the operators: it is the operator who has used the parts in a particular container who takes the card and sends it up-stream; it is the truck driver who delivers it; it is the operator further up-stream who receives the card and decides to act on it. That is, due to the operators’ control of the execution of the *kanban* protocol, the direct coupling of the *kanban* system to the field of work can be severed whenever they deem it appropriate to exercise that control.

An artifact is, of course, more than a permanent symbolic construct; it has a specific material format which, in itself, is of importance to its use.¹ For example,

¹ In his analysis of cognitive artifacts, Norman introduces a distinction between ‘surface representation’ and ‘internal representation’ which is roughly equivalent to our distinction between the artifact and its material form on one hand and the protocol on the other hand: “Some artifacts are capable only of a surface level representation. Thus, memory aids such a paper, books, and blackboards are useful because they allow for the display and (relatively) permanent maintenance of representations. […] These devices are primarily systems for making possible the display and maintenance of symbols. They implement the ‘physical’ part of the physical symbol system. These are called surface representations.
consider the simple checklist again. The checklist can be conceived of as an artifactually imprinted protocol that has been deliberately and carefully designed to reduce local control, typically in safety-critical environments. The use of the checklist requires that the actor employs a strategy for sequential execution which permits him or her to ensure that the steps are done in the correct order and that each step is done once and only once. The material format of the checklist as an artifact may be of assistance to the actor in ensuring this:

‘The fixed linear structure of the checklist permits the user to accomplish this by simply keeping track of an index that indicates the first unexecuted (or last executed) item. Real checklists often provide additional features to aid in the maintenance of this index: boxes to tick when steps are completed, a window that moves across the checklist, etc.’ (Hutchins, 1986, pp. 47 f.; cf. also Norman and Hutchins, 1988, p. 9)

In a similar vein, in his discussion of the specific affordances provided by the material format of written text, Jack Goody observes that writing introduces certain spatio-graphic devices such as lists, tables, matrices by means of which linguistic items can be organized in abstraction from the context of the sentence (Goody, 1987) and points out that the spatio-graphic format of an artifact can stipulate behavior by reminding an actor of items to do and directing attention to missing items: ‘The table abhors a vacuum’ (Goody, 1987, p. 276). This is, again, eloquently illustrated in the case of the bug form.

**Proposition 14.** The material format of the artifact conveys stipulations to the articulation of distributed activities in ways that appropriately reflect salient concerns of the protocol.

By way of concluding this discussion, it is important to keep in mind that an artifact only conveys stipulations within a certain social context, within a certain community, in which the protocol and any change to the state of the protocol have a (more or less) certain and agreed-to meaning and that it only does so under conditions of social accountability. The point we want to make here, however, is that the specific structural and behavioral properties of the artifact (its material formats as well as its protocol, if such have been incorporated in the artifact) are formed to serve the purpose of conveying specific stipulations within this particular context by constraining and forcing the actors’ behavior.

### 3.3. Coordination mechanisms: mutual alignment

Consider, again, the case of software testing in the S4000 project. As observed previously, in addition to the bug report mechanism, the software designers...
introduced and used a variety of other artifactually embedded protocols to handle the complexity of coordinating the distributed testing.

For example, a project schedule in the form of a spreadsheet was used to capture and display the relationships between actors, responsibilities, tasks, and deadlines. When using the bug report mechanism, participants would consult the project schedule to obtain information about which actor would be the ‘platform master’ responsible for verifying the presumably corrected bug; this was implicitly indicated in the bug report form by the number of the platform period. However, a platform period number indicated in the bug report form also implicitly indicated a deadline for the correction task to be finished. Moreover, this deadline was not explicitly stated but was again inferred from the reference to the platform period or, in other words, through a ‘subscription’ to the project schedule spreadsheet where the deadline would be stated explicitly. That is to say, from the point of view of the involved coordination mechanisms, one coordination mechanism (the bug report mechanism) subscribed to the specification of a role to be provided by another mechanism (the project schedule).

It is worth noticing that such concerted interactions between coordination mechanisms makes it possible for the instantiation of a mechanism (e.g., a particular bug report) to be executed while still not completely specified; the missing specifications can be ‘filled-in’ later by consulting another mechanisms. That is, this example suggests that coordination mechanisms can subscribe not only to role specifications but to definitions or specifications of any variable or attribute. Because of this, actors do not necessarily need to specify explicitly what can be inferred from other mechanisms at some point in time.
The case also demonstrates a more active form of interaction between coordination mechanisms, namely in the form of one mechanism inciting the execution of another coordination mechanism when a certain condition occurs (see figure 3). For example, when a reported bug was accepted as a bug, a new task was announced and entered in the project schedule, i.e., the change in the state of one mechanism (the bug report form) triggered operations on data structures of another mechanism (the project schedule). Similarly, when a bug was reported to have been corrected, yet another task was announced, namely the task of verifying the correction. However, this task was not initialized until the ‘platform master’, who would be responsible for the verification, had been appointed and until the point in time where the next integration period was going to start had occurred. This starting point was specified in the project schedule.

Hence,

**Proposition 15.** Coordination mechanisms are specialized constructions that are devised to support certain aspects of the articulation of a specific category of distributed activities within a particular cooperative work arrangement and the use of a coordination mechanism may therefore require that it is aligned with other mechanisms devoted to different aspects of the articulation of those activities or to related activities.
4. Computational coordination mechanisms

As noted above, coordination mechanisms based on paper artifacts (e.g., forms, catalogues, time tables) have been around for ages and they are used on a massive scale in modern work settings. While mundane and unassuming, they have crucial affordances: (a) the artifact can represent and convey stipulations among actors in a permanent and publicly accessible form; (b) the protocol and the artifact can be defined and specified by the actors themselves — operators, clerks, managers, auditors, etc. — by means of the ordinary skills of their professions; (c) actors have total control of the interpretation and execution of the protocol and can, under conditions of social accountability, modify or deviate from the protocol; (d) the artifact can dynamically represent state changes to the protocol and mediate these among actors, and (e) multiple coordination mechanisms can be aligned, seamlessly and smoothly, by actors.

Nonetheless, such mechanisms have serious inherent limitations: (a) state changes to the protocol are conveyed by paper and similar unwieldy artifacts and the speed and pattern of propagation of changes to the state of the protocol are thus severely limited; (b) the protocol is only immediately visible to actors to the extent that the protocol is mapped onto the symbolic construct of the artifact that serves as an intermediate; (c) modifications to the protocol only take effect when or if actors become aware of them through other channels; (d) maintaining a conventional, paper-based coordination mechanism involves a plethora of mind-numbing operations; (e) it involves massive housekeeping efforts and it may thus be practically impossible for actors to obtain an overview of the state of the protocol, and (f) the seamless and smooth alignment of multiple coordination mechanisms is only feasible to the extent that the same actors are involved with the coordination mechanisms in question.

These limitations with conventional coordination mechanisms become increasingly problematic as modern industrial, service, and administrative organizations need to be able to operate in a radically flexible and adaptive and yet highly coordinated fashion. In view of these issues, it seems obvious to explore whether it is possible to construct computational coordination mechanisms in which the allocation of functionality between actor and artifact is changed in such a way that aspects of the protocol are incorporated in a computer artifact in a computational form. Accordingly, in a computational coordination mechanism the artifact plays a dual role: it still serves as an artifact in the narrow sense of a permanent symbolic construct, of course, but in addition, the computational coordination mechanism as a software artifact incorporates a computational protocol that operates on the artifact as a permanent symbolic construct:

*Proposition 16.* A computational coordination mechanism can be defined as a software artifact in which the artifact in the sense of a permanent symbolic construct as well as (aspects of) the protocol are incorporated in such a way...
that changes to the state of the protocol induced by one actor are conveyed by the computational artifact to other actors according to the protocol.

Now, since coordination mechanisms are local and temporary closures (Gerson and Star, 1986), no computational coordination mechanism will be able to handle all aspects of articulation work in all work domains. Particular computational coordination mechanisms will be designed to support cooperating actors in specific aspects of their articulation work which are particularly complex and which, most likely, are specific to the given work domain. A computational coordination mechanism should thus be conceived of as a specialized software 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.

Proposition 17. A computational coordination mechanism should be conceived of as a specialized software device incorporated in a particular software application so as to support the articulation of the distributed activities of multiple actors with respect to the field of work as represented by the data structures and functionalities of that application.

Based on the analysis of the empirical studies of artifactually imprinted protocols, a set of requirements for computational coordination mechanisms can be identified. The requirements can be organized into two categories: ‘malleablility’ and ‘linkability’.

4.1. Malleability

Since coordination mechanisms are ‘resources for situated action’ (Suchman, 1987), a computational coordination mechanism must be malleable in the sense that users are supported in defining its behavior.

Organizational demands and constraints change, and procedures and conventions changes accordingly. It should thus be possible for actors to design and develop new computational coordination mechanisms 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:

Proposition 18. A computational coordination mechanism must be constructed in such a way that actors can define and redefine the protocol of the mechanism so as to be able to meet changing organizational requirements.

On the other hand, in view of the inexorably contingent nature of work, actors must be able to make local and temporary changes to the protocol, for instance by
suspending or overruling a step, by ‘rewinding’ a procedure, by escaping from a situation, or even by restarting the mechanism from another point. For example, as noted previously, the bug form protocol was respecified during its execution as new designers became involved, as erroneous classifications of bugs were discovered, as designers rejected the responsibility ascribed to them, etc. In other words, actors must be able to exercise local control over the execution of the mechanism.

More generally stated, the specification of an already defined protocol should not be conceived of as a solitary act of creation. As stated in proposition 9, a coordination mechanism is in principle under-specified (Suchman, 1983; Suchman and Wynn, 1984; Suchman, 1987). Thus, a protocol will, at least to some extent, typically be specified incrementally in the course of the work, while it is executed. A protocol and the whole equipage of ensuing conventions can be invoked implicitly, without any explicit announcements, for instance by certain actors taking certain actions (Strauss, 1985; Schäl, in press). 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 coordination mechanism to a larger degree than deemed necessary, a computational mechanism must be constructed in such a way that a partial specification of the protocol is possible. That is, it should be feasible for attributes of the protocol specification to be left un-specified and for the missing specification to be provided, at a later stage, perhaps by another mechanism or by inference from actions taken by actors. For example, if actor A starts performing task $a$, he may then be committed to accomplish task $a$ and it may also be inferred that he has assumed the role $x$ defined as responsible for task $a$.

Hence,

**Proposition 19.** A computational coordination mechanism must be constructed in such a way that its behavior, at least partially, can be continually respecified while it is being executed so as to allow for incomplete specification of the protocol and for local and temporary modifications of its behavior to cope with unforeseen contingencies.

From these requirements (proposition 18-19) it follows that the definition and specification of the protocol must be ‘visible’ to actors, not only in the sense that it is accessible but also, and especially, that it makes sense to actors in terms of their articulation work:

**Proposition 20.** In order for actors to be able to control the execution of the mechanism and redefine the protocol, the protocol must be visible, i.e., accessible and intelligible, to actors at the semantic level of articulation work.
We are not here addressing the issue of which modality of presentation is most appropriate: graphs, trees, nets, matrices, or standardized prose. The point is that the protocol must be visible at a semantic level, at a level of granularity, and in a modality which is appropriate for the specific work domain at hand. That is, the objects and functional primitives available to actors for defining or specifying the protocol must be expressed in terms of operations of articulation work with respect to constructs such as roles, actors, tasks, activities, conceptual structures, resources, and so on that are meaningful to the participants involved in terms of their everyday work activities.

Moreover, as a specialized software device incorporated in an application so as to support the articulation of distributed activities with respect to the field of work as represented by the data structures and functionalities of that application, a computational coordination mechanism must be distinct from the other software components of the application in which it is incorporated (proposition 5) in order for the mechanism to be malleable to actors at the semantic level of articulation work. If the coordination mechanism cannot be defined and specified independently of the other components of the system, malleability cannot be bounded and actors will thus be confronted with a vast space of possibilities at innumerable semantic levels that will lead to much confusion.

On the other hand, articulation work is always fundamentally conceived of with respect to the common field of work and in terms of the specific ordering of objects and processes constituting this field of work. The bug report protocol, for example, refers to entities of the field of work such as ‘module name’, whereas the kanban protocol refers to such entities as ‘part name’ and ‘number of parts’ etc. Accordingly, a computational coordination mechanism must be constructed in such a way that its stipulations can be related to and expressed in terms of the objects and processes of the field of work. For example, a computational coordination mechanism incorporated in a collaborative-writing application, for instance to support the coordination of the flow of distributed activities of writing, editing, evaluating, reviewing, proofreading, and accepting the contributions to a technical report, would need to be able to relate to the usual data structures of the word processor part of the application: text strings, formatting instructions, document components (paragraphs, sections, headings, tables, headers, footnotes, etc.).

**Proposition 21.** A computational coordination mechanism must be constructed in such a way that actors can relate the definition and specification of the mechanism to pertinent entities in the field of work as represented by the data structures and the functionality of the application in which it is embedded.

Since articulation work, as we noted earlier, is a recursive function (Gerson and Star, 1986), changing a coordination mechanism may itself be done cooperatively,

---

1 Nor are we excluding other sensory modalities than vision.
as part and parcel of the cooperative effort. That is, it should be possible to change the coordination mechanism while it is running, without having to suspend all activities within the cooperative ensemble for some time. Moreover, changing a coordination mechanism (permanently or temporarily) may itself be a cooperative activity which may need to be supported:

*Proposition 22.* Since a computational coordination mechanism must be malleable, it must be constructed in such a way that actors can *control the propagation of changes to the protocol.*

### 4.2. Linkability

Coordination mechanisms are local and temporary closures, as we have frequently noted. Thus, a given cooperative work arrangement will — in all but the most extreme circumstances — be working with multiple CSCW applications with different coordination mechanisms which are interrelated. For example, in the domain of engineering design, coordination mechanisms supporting the articulation of distributed activities may be incorporated in project management tools, CAD tools, and process planning tools as well as in generic ‘groupware’ tools such as departmental calendar systems and collaborative writing tools.

In the case of paper-based coordination mechanisms, the artifact only statically reflects the protocol in the form of the various fields representing salient concerns of the protocol; the artifact is completely inert and any changes to the state of the protocol or the artifact are exclusively the result of actions by human actors (even when, as in the *kanban* case, it may appear the result of a monstrous, distributed machinery). Because actors are totally involved in the execution of paper-based mechanisms — they are completely ‘in the loop’ of articulation work —, they are also relatively well-situated to align the different coordination mechanisms with respect to each other, at least in so far as the different actors generally are equally involved in the use of the different mechanisms. When the allocation of functionality between artifact and actor changes, however, as a result of the construction and introduction of computational coordination mechanisms, the ability of actors to align coordination mechanisms in the former intuitive way deteriorates.

Thus, as coordination mechanisms are incorporated in different applications and their functionality is enhanced, users will be inundated with overhead activities of aligning the different mechanisms: updating each mechanism with respect to changes to other mechanisms. In order not to create an impedance between the multitude of interlaced — individual and cooperative — activities with respect to the multiple applications required to do the work in a particular setting, it should be possible for actors to link the coordination mechanisms incorporated in the various applications so as to facilitate the fluid interrelation of articulation work with respect to these applications. Furthermore, since cooperative work is articulated with respect to and in terms of domain-specific objects
and structures, coordination mechanisms incorporated in applications must provide access to these objects and structures, e.g., via information systems providing access to common repositories (previous designs, components, work in progress, drawings, patents), to other available human and technical resources (skills, machinery), to statutory constraints, and so on.

Thus, a central problem in the design of CSCW facilities that are actually able to support real-world cooperative ensembles in handling the increasing complexities of their work, is to provide an appropriate ‘interface’ to the wider organizational context as represented by other CSCW facilities incorporated in other applications as well as domain-specific information systems such as MIS, OIS, CIM, and CASE systems (De Michelis and Grasso, 1993; Fuchs and Prinz, 1993; Prinz, 1993). The challenge is, as Ellis and Keddara aptly put it, to make groupware ‘organizationally aware’ (Ellis et al., 1995).

**Proposition 23.** Computational coordination mechanisms should be constructed in such a way that they can be linked to other coordination mechanisms in the wider organizational field.

Since computational coordination mechanisms must be able to interact in a concerted fashion, they must be constructed by means of the same set of elements, at the same semantic level. In order to ensure that, a general notation for constructing computational coordination mechanism is required.

**Proposition 24.** In order to ensure comprehensive linkability of computational coordination mechanisms, a general notation for constructing computational coordination mechanism is required.

Now, malleability and linkability are evidently contradictory requirements, since the former hinges upon the possibility of modifying the behavior of a mechanism and the latter upon the stability of the behavior of other mechanisms. This conflict is unavoidable since ‘no representation of the world is either complete or permanent’ and each mechanism therefore is a ‘local and temporary closure’ (Gerson and Star, 1986). Nevertheless, the conflict can be alleviated in different ways: Basically, coordination mechanisms should be made tolerant of limited and relatively trivial modifications of other mechanisms by means of interface agents. For example, when the modifications of a particular coordination mechanism affect its alignment to other mechanisms in such a way that the interactions become unfeasible (e.g., because of a loss of synchronization) or generate unwanted effects (e.g., deadlocks), a solution can be based on the introduction of suitable mediating agents which serve as “clutch” between the mechanisms (cf. Genesereth and Ketchpel, 1994).

However, when modifications are too radical to be handled by such interfaces, and the problem therefore recurses (Bowker and Star, 1991), the different cooperative ensembles of course have to sort out the mess and negotiate a new arrangement. In line with the recursive nature of articulation work, such
negotiations may themselves be governed by suitable coordination mechanisms. This suggestion is not the fruit of idle speculation on our part but is grounded in the field study evidence. For example, in a study the cooperative production of technical documentation in a Danish manufacturing company\(^1\) we have observed cases where one coordination mechanism, a ‘construction note’, is used to govern the distributed process of negotiating proposed changes to another coordination mechanism, namely the ‘classification scheme’ that is required to govern the distributed production and dissemination of the massive amounts of technical documentation in the company (Schmidt et al., 1995). In other words, the problem recurses, but so does the solution.

5. Related research

The aim of incorporating preconceived protocols in computer systems so as to support the articulation of distributed activities is, of course, shared by many researchers in the CSCW community and a number of notations have been proposed on the basis of more or less explicit and elaborate models of articulation work.

A very 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 the integration of many types of information and applications and at the same time give users the ability to tailor this integration by providing a set of basic primitives such as objects, views, agents, and links. The notation provided by OVAL is very abstract and flexible and is thus an important contribution to the definition of a general framework for the construction of coordination mechanisms. However, OVAL is problematic in that the basic primitives provided by its notation are not at the semantic level of articulation work.

By contrast, the various CSCW applications incorporate coordination protocols which are defined at a higher semantic level than in OVAL. However, this advantage is bought at a price, namely at the cost of malleability. In fact, some of these applications are experienced as excessively rigid, either because the protocols incorporated in them are not accessible to the actors and cannot be changed at all, e.g., THE COORDINATOR (Winograd and Flores, 1986; Flores et al., 1988), or because the facilities for changing the protocols do not support the actors themselves in redefining the mechanism, at the semantic level of articulation work, e.g., DOMINO (Kreifelts et al., 1991a; Kreifelts et al., 1991b).

These CSCW applications can be viewed as representatives of the first generation of the new class of workflow management systems which, in recent years, has received much attention. Without attempting any detailed discussion of this rapidly growing class of systems, we want to point out that they all share some general characteristics. First of all, none of them observes the explicit distinction between articulation work and the field of work with the consequence

\(^1\) This study was conducted by Hans Andersen and reported in (Andersen, 1995).
that the malleability of the incorporated computational protocol becomes problematic (cf. proposition 17). Moreover, they are all based on partial models of articulation work due to the fact that each system proposes a set of basic objects of articulation work which is determined by the particular approach they take to modeling workflows. For example, some of them are focused on tasks and roles, others are more focused on resources and on their flow across actors. In any event, none of them appreciates the crucial role of ‘the artifact’. In fact, no workflow management system supports actors in constructing, for instance, a bug report or a binder with all their awareness capabilities, but just the bug handling protocol for the allocation of tasks (basically, construction of to do lists) and for managerial control. They thus incorporate predefined data structures and functionalities devoted to defining protocols and to governing the (automatic) assignment of tasks to actors. Since their emphasis is more on managerial supervision of cooperative activities than on horizontal coordination, the additional features built on top of this information usually consist in simulating the workflow’s behavior under different workloads and in reporting performance indices concerning the accomplishment of the tasks constituting the workflow.

A number of recent research projects aim at making CSCW applications malleable at the level of articulation work by supporting actors in the evolutionary design of the incorporated coordination mechanisms.

EGRET (Johnson, 1992) supports cooperative work in a dynamically changing environment by giving users the ability to develop new ‘schemes’ which can include tasks, descriptions, names, etc. EGRET keeps track of the variation in the schemes by providing facilities for listing deviations from each schema. This enables all group members to see, for example, the possible 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 schemes. Some degree of local control of execution of changes is supported and all changes are propagated throughout the database.

Strudel (Shepherd et al., 1990) is based on the notion of task-move structures encompassing conversations, tasks, and notifications. These structures can be flexibly defined during their activation on the basis of some linguistic cues for specifying the next move. As in OVAL, each move is defined by the users as a semi-structured construct containing, among the domain-dependent ones, an attribute which specifies the set of next possible moves. At execution time, the next move can be selected from this set or constructed ‘on the fly’ by the user.

The ConversationBuilder (Kaplan et al., 1992; Bogia et al., in press) 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’. On the other hand, policies can be built out of mechanisms coping with interdependencies among tasks and sharing of data. That is, the ConversationBuilder shares an approach similar to Strudel but enriches its basic mechanisms. While the Obligations model (Bogia and Kaplan, 1995)
implemented on top of the ConversationBuilder imports most of its ‘mechanisms’, it focuses on the issue of the management of multiple versions that is crucial to evolutionary design. These features allow one to organize the various alternatives in a multilayered structure so that they can be selected and composed in a dynamic and flexible way. The various levels can be associated in a natural way to different degrees of visibility according to different criteria, e.g., according to the levels of the organizational structure.

Similarly, Regatta (Swenson, 1993; Swenson et al., 1994) proposes a ‘collaboration model’ for the design of business processes addressing requirements which, in our terms, can be rephrased as visibility, malleability, and, to some extent, control of the interface to the field of work: ‘the model should anticipate the use of external tools and still provide sequencing support of the various tasks that need to be done with these tools’ (ibid., p. 19).

While far from exhaustive, this discussion shows a clear trend within CSCW toward the development of malleable and linkable computational coordination mechanisms. It is noteworthy that almost all of the more advanced systems have been designed in response to the needs and demands of real-world settings and have tested in such settings. Each of them propose very interesting strategies for satisfying some of the requirements. However, none of them is able to satisfy all of them. Our ambitious goal to define a general notation should be thus viewed as an appreciation of the related efforts which we accordingly attempt to locate within a comprehensive framework as well as an attempt to satisfy the requirements not addressed by the related efforts.

6. Requirements for a general notation

From the definition of computational coordination mechanisms (proposition 4) and from the exposition of the general requirements for computational coordination mechanisms (proposition 18-23), we can derive some specific requirements for a general notation supporting their construction.

A computational coordination mechanism — in short, a C₂M — has been defined (proposition 16) as a computer facility that incorporates the artifact as well as (aspects of) the protocol of a coordination mechanism. This definition allows one to express the relationship between a coordination mechanism and its computational part in terms of an ‘incorporation’ that makes computational aspects interleave ‘in a continuum’, so to speak, with aspects that cannot be represented within a computer. Accordingly, the notation must be open to represent these non-computational aspects as specific points in the control flow where the context (either the user or other applications) should be invoked so that it takes the control up to the point where control can be resumed by the C₂M. In so far as it is able to do so, the notation resolves two well known problems in systems development. Firstly, the crucial decision about the allocation of functionality between actors and system which we discussed previously, in the
motivations for the notion of coordination mechanism (proposition 4). Secondly, 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 practitioners is a major source of misconceptions, misunderstandings, and misinterpretations in systems development and therefore one of the most frequent reasons of systems failure. In fact, these classes of practitioners do not generally share the same skills and perspectives regarding the system and the setting where the system is expected to be put to work.

The smooth transition from the description of a coordination mechanism into the definition of its computational incarnation is one of the preconditions for the solution to the two above-mentioned problems in the evolutionary construction of C²Ms.

Proposition 25. The notation should allow for a smooth transition from the description of a coordination mechanism that has been identified in an analysis of a cooperative ensemble to the definition of the corresponding C²M.

Furthermore, the definition of a C²M as a computer artifact able to convey information about changes to the state of the protocol implies the following requirement to the notation:

Proposition 26. The notation should be able to express, at any level of granularity, which notifications should take place in the presence of certain conditions or internal states of the various components of the C²M.

An additional set of requirements for the notation derive from the issues of malleability and linkability required of C²Ms, as discussed in section 4. Let us consider all of them in turn. First of all, malleability is required in any phase of the C²M life-cycle (propositions 18-19).

Proposition 27. The notation should support the manipulation of the definition and specification of the C²M, i.e., in the form of permanent modifications as well as in the form of local control of execution.

Moreover, since visibility is a necessary condition for malleability (proposition 20), the notation should be accessible, i.e., understandable and available, to all the actors involved in the definition and specification of the C²M. Accordingly, the notation has to be easily tailorable to different actor needs which might depend on the state of the C²M as well as on the particular actor’s role and skill, and provide them with a support for the implementation of the changes (proposition 22).

Proposition 28. The notation should be accessible by the actors, where actors may be end-users, experts on organizational design and process
coordination, as well as to people providing technical environments for the development of C2Ms.

The notation should support actors in controlling the propagation of changes to the mechanisms, not only by notifying the involved actors of modifications to the protocol but also by enabling them to anticipate the effects of modifications so as to choose the most appropriate strategy for the propagation of changes.

**Proposition 29.** The notation should allow for the definition of strategies for controlling the propagation of changes.

The requirement to allow for partial specification implies that the notation should provide facilities that make the C2M function even if not all the related pieces of information are available (proposition 22). In particular, C2Ms — as almost all computer artifacts — cannot be constructed under the assumption that the behavior of users is completely mediated by the C2M. Indeed, it is most likely that some actions or negotiations pertinent to the articulation work are performed outside the system and that the latter is requested to guess and reconstruct them on the basis of their consequences. For example, an assignment of an activity to an actor can happen in a face-to-face conversation and can be followed by the allocation of the related resources to the actor; the assignment should be inferred when the actor starts the activity by possessing all the access rights to the required resources.

**Proposition 30.** The notation should allow for the definition of methods for deriving from the current state of the C2M the missing pieces of information just at the point when they are required for continuing its execution.

Finally, linkability (proposition 23) requires the identification of a set of necessary and sufficient means to let C2Ms interact at the semantic level of articulation work.

**Proposition 31.** The notation should provide support for the definition of the required interactions across C2Ms.

### 7. Ariadne: A notation for malleable and linkable coordination mechanisms

Ariadne is a notation for the construction of C2Ms that meets most of the above requirements by virtue of the following basic properties.

The studies of how procedures and artifactually imprinted protocols are used by actors in everyday work activities led to the identification of a minimal set of *objects of articulation work* and of *elemental operations* on these objects (proposition 20), as shown in the table of figure 4. It is presumed 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 and the model is therefore open to modifications. A similar idea of selecting objects and related operations has been suggested by Malone and others (Malone and Crowston, 1990) as an initial foundation for an interdisciplinary ‘coordination theory’.1

The identified objects and operations constitute the articulation work model underlying Ariadne. In this model objects and operations are ordered along two dimensions of articulation work. On the one hand articulation work is distinguished according to its status, that is, nominal and actual (proposition 9); on the other hand the model distinguishes between articulation work with respect to the cooperative work arrangement and the field of work.

The first dimension, the distinction between the nominal and actual statuses, identifies the objects and operations that are pertinent to the definition and specification of the C2M, respectively. This distinction highlights the fact that the transition from the nominal to the actual status is not merely a refinement, since the nature of the involved objects and of the related operations are different. By way of illustration, an activity denotes a work process as an unfolding course of action in terms of those aspects of a work process that are relevant to doing the work with the currently available resources, whereas a task denotes an operational intent, irrespective of how it is implemented (Andersen et al., 1990); in other words, a task is expressed in terms of what, an activity in terms of how. That is, a task is accomplished while an activity ceases.

The other dimension highlights the distinction between objects pertaining to the cooperative work arrangement addressed by the C2M at hand and objects representing what is acted upon, i.e., the field of work. Notice that ‘human resources’ belong to the former so as to indicate that any action on them would imply a change of the level of recursiveness (proposition 2). In addition, the set of objects and operations pertaining to the field of work contains, together with the more obvious resources of various kinds, an item termed ‘conceptual structures’, that is, the modeling constructs needed to express the conceptualizations of the field of work (definitions, classifications, etc.) that the participants of the cooperative ensemble have adopted in order to be able to refer to the multifarious objects and processes of their common field of work.

The objects of the articulation work model are the basic building blocks made available by Ariadne whereas the elemental operations are used to identify, and to define the meaning of, the attributes characterizing the objects and relations among them (the diagram of figure 5 focuses on the relations while the attributes are described in detail in (Simone et al., 1995a)).

---

1 This effort later evolved into the current attempt to define ‘tools for inventing organizations’ (Malone et al., 1993).
<table>
<thead>
<tr>
<th>Nominal</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objects of articulation work</strong></td>
<td><strong>Operations with respect to objects of articulation work</strong></td>
</tr>
<tr>
<td><strong>Articulation work with respect to the cooperative work arrangement</strong></td>
<td></td>
</tr>
<tr>
<td>Role</td>
<td>assign to [Committed actor]; responsible for [Task, Resource]</td>
</tr>
<tr>
<td>Task</td>
<td>point out, express; divide, relate; allocate, volunteer; accept, reject; order, countermand; accomplish, assess; approve, disapprove; realized by [Activity]</td>
</tr>
<tr>
<td>Human resource</td>
<td>locate, allocate, reserve;</td>
</tr>
<tr>
<td><strong>Articulation work with respect to the field of work</strong></td>
<td></td>
</tr>
<tr>
<td>Conceptual structures</td>
<td>categorize: define, relate, exemplify relations between categories pertaining to [Field of Work];</td>
</tr>
<tr>
<td>Informational resource</td>
<td>locate, obtain access to, block access to;</td>
</tr>
<tr>
<td>Material resource</td>
<td>locate, procure; allocate, reserve to [Task];</td>
</tr>
<tr>
<td>Technical resource</td>
<td>locate, procure; allocate, reserve to [Task];</td>
</tr>
<tr>
<td>Infrastructural resource</td>
<td>reserve;</td>
</tr>
</tbody>
</table>

**Figure 4.** The ‘articulation work model’: The table identifies the elemental objects of articulation work and elemental operations on these objects.
The provision of a notation at the semantic level of articulation work is one of the basic characteristics of Ariadne which distinguishes it from many proposed environments for the development of CSCW applications. As noted in section 5, these environments are based — in the most advanced proposals — either on partial models of articulation work or on languages at the semantic level of the manipulation of general-purpose objects that actors must specialize in order to raise them to the semantic level appropriate for building C²Ms supporting articulation work. This transformation involves the additional effort of defining an ad hoc articulation work model and often leads to the definition of partial articulation work models, since it is part of the development of a specific application. The ad hoc and partial character of these models is a source of problems when the resulting C²Ms are to be modified and linked in the course of the distributed and evolutionary construction process. By contrast, Ariadne provides a finite and expressive framework for developing a shared understanding across the activities of the distributed and evolutionary construction process. In this framework, the model of articulation work plays the fundamental role governing the design of C²Ms, of allowing for their malleability and linkability, of governing the impact of changes, of supporting the handling of partial specifications and, finally, of making the notation visible to and hence usable to all categories of actors.
Proposition 32. The expressive power of Ariadne is determined by the fact that the notation is at the semantic level of articulation work and that Ariadne consequently contains the linguistic features necessary to express the objects of articulation work together with their relations, to build the protocols and artifacts on top of them, to manage the transition from the nominal to the actual status of C²Ms, and to express the relationships of each C²M to its field of work and to the cooperative work arrangement it supports.

Even if Ariadne is founded on a small selection of basic elements (the objects and their relations, describing the dynamic aspects of the protocols), its richness and usability is guaranteed by the flexibility by which the various elements can be combined to define and link different protocols. The granularity determined by the semantic level of articulation work guarantees that the required flexibility is manageable — at the same ‘semantic’ level — by actors involved in the incremental construction of C²Ms. Indeed, to become operational the semantics given by the articulation work model has to be supported by a syntax whose elements have associated a univocally defined behavior coherent with the model. Moreover, since these elements have to be composed to obtain both richness and usability of the notation, their syntax and operational semantics have to be both formally defined and compositional. On the one hand, formality allows one to univocally define interfaces between the operational semantics of the various elements corresponding to the objects of the articulation work model and the protocols built on top of them. On the other hand, Ariadne’s compositionality is just based on, and governed by, these interfaces.

The compositionality of the notation and the concomitant rules of composition allow actors to manage modifications in a sound way since they, in addition to the formal semantics, can support consistency checks and provide tools for evaluating the impact of modifications (proposition 29). Moreover, modifications can be achieved more easily and less expensively if the elements of the notation can be isolated and substituted and the impact of the changes on other elements thus can be reduced. Together with compositionality, Ariadne provides animation capabilities and has been prepared for incorporating simulation capabilities and techniques for governing changes and assessing their impact (e.g., Bogia and Kaplan, 1995; Ellis et al., 1995).

Proposition 33. Ariadne possesses a formal and compositional syntax and operational semantics. These are the pillars on which the support for the construction of C²Ms is based, both in their nominal and actual status.

As for the possibility of composing C²Ms and their constituent components (proposition 31), a comparative analysis of field study findings led to the identification of three basic modes by which C²Ms can be linked. According to the above considerations, each mode has a formally defined operational semantics associated to it.
In subscription mode a C2M makes the behavior of another mechanisms part of its own behavior. For example, the subscription could involve the activation of another CM devoted to manage special types of negotiations or classification schemes as a way to access the resources of the field of work; or the subscription could activate a policy associated to an object of the articulation work, namely roles, actors, resources referred to by the current C2M.

In inscription mode a C2M provides information about its current state to another C2M (or, conversely, a C2M obtains information about the current state of another C2M). The inscription can be done in two ways: the reaction by the target C2M can be either compulsory or voluntary. On the one hand, in the case of compulsory inscription, a C2M can write into the space of the target C2M in order to provide information that is necessary for the behavior of the target C2M (or, conversely, a C2M can read from the space of another C2M). In this case, the target C2M is expected to react accordingly and, if does not do so, then the operational semantics of compulsory inscription mode has to incorporate time-outs and solicitation in order to reduce the risk of (partial) blocking of the involved C2Ms. On the other hand, in the case of voluntary inscription, a C2M tells another C2Ms of some events or conditions and thus makes the other C2M ‘aware’ of these events or conditions (or, conversely, a C2M asks about another C2Ms of some events or conditions). In this case, the reaction of the target C2M is voluntary in that it is provided with morsels of information that are supplementary to what is imperative. That is, the operational semantics of compulsory inscription mode has to incorporate capabilities to allow the target C2M to voluntarily filter the provided information (Malone et al., 1987; Gasparotti and Simone, 1990; Fuchs et al., 1995).

In prescription mode a C2M overwrites the definition of the target C2M’s behavior. This interaction mode allows Ariadne to honor the recursive nature of articulation work (proposition 2). In fact, in the prescription mode a given C2M can change the (nominal) definition of another C2M, that is, the definition of its protocol, or its (actual) specification, for example, by enforcing a special state during its execution.

Subscription and inscription modes apply also to the C2M’s components (objects of articulation work, artifact, protocol) to express their interactions. For example, an object of articulation work, for instance, a resource, can be accessed in the subscription mode to activate the policies governing its usage. The compulsory inscription mode typically expresses the reading from and writing to the artefact by the protocol in order to acquire and make visible imperative information. Finally, the voluntary inscription mode is typically used by the artefact to convey awareness of its internal changes to the protocol. To the contrary, prescription mode applies just to the whole C2Ms since it expresses the cooperative handling of the modifications with the single exception of two objects of articulation work, namely role and actor. In fact, in this case the handling of the modifications can be seen as a degeneration of the previous case where the protocol and the related artefact collap to a single person behaviour.
We are not claiming that these modes are fully original: they can be related, for example, to well-known features in programming languages (e.g., call by reference, co-routines, reflective features). The point is that these three modes express the capabilities provided by the more advanced programming languages at the semantic levels of articulation work and that the combined use of them is able to represent all the needs of linking C2Ms as identified in our studies (up to now).

Collectively, these modes constitute the skeleton of an Interoperability Language (Simone et al., 1995b) that is part of Ariadne as a feature available to actors in the evolutionary construction of a C2M in order to link C2Ms with other mechanisms in the wider organizational context. Moreover, the Interoperability Language is the basic means to define the interfaces between the elements which are the basis of the compositionality of Ariadne. That is, the interoperation of coordination mechanisms and of their components is described in a uniform way and, most importantly, is determined by the semantic level of articulation work.1

This uniformity makes it possible to conceive of a flexible allocation of functionality between actors and C2Ms.

**Proposition 34.** Ariadne contains features to express the various modes in which C2Ms and their components interoperate.

Finally, the variety of elements constituting the notation, the different modality in which they are used in the nominal and actual status of the C2M, and the requirements of malleability and linkability suggest to provide the notation with an internal structure organized into three levels that are called α, β, and γ, as illustrated in the central part of figure 6.

---

1 This goes exactly the other way round with respect to the trend of applying programming language concepts to non-programming activities, like e.g., object-oriented approach.
Two are the motivations leading to the identification of the $\gamma$-level. First of all, the aim of making it possible to define the language to be used to construct the C$^2$Ms. This aim tries to overcome a limitation of workflow systems, and in general of CSCW applications, mentioned in section 5, namely the fact that they impose a specific modelling approach. For example, the dynamic aspects of the protocols, are described in most applications and environments through partial models of articulation work of which the most common are focused on the flow of objects across organizational units (individuals, tasks, etc.), on the flow of control across actions, on some communication patterns among roles (negotiation), or on some predefined combination of these (Ellis, 1979; Cook, 1980; Shepherd et al., 1990; Medina-Mora et al., 1992; Swenson et al., 1994).

In recognition of the fact that the adequacy of the language depends strongly on the aspects of articulation work that are being addressed and on the organizational environment in which the coordination mechanism is intended to operate, Ariadne does not impose a preconceived modeling approach. Rather, by exploiting its compositionality, Ariadne allows designers selecting the interpretation to be associated to the formal structures adopted to construct C$^2$Ms out of the multifarious interpretations afforded by the articulation work model. This functionality identifies a first level, where the grammar

The second reason for identifying a $\gamma$-level of the notation is the requirement of linkability of C$^2$Ms. In fact, there is a need of a point where the Interoperability Language is made available to the designer who uses its features as an integrated part of the languages for constructing C$^2$Ms, and this point is exactly where this language is defined. Let us notice that, as a special case, the various linking modes incorporated in the Interoperability Language can be exploited to let C$^2$Ms described with different languages interoperate. To our knowledge, this is an original way to look at interoperability of heterogeneous CSCW applications through their coordination mechanisms.

The motivations leading to the remaining levels, namely the $\beta$-level and the $\alpha$-level, are related to the necessity of account for the distinction between the nominal and actual status of the C$^2$Ms. In fact, each status identifies different primitives the actors utilize to make changes in the C$^2$Ms according to the malleability requirement: these primitives are different since different is the status of the objects manipulated in the two levels. The primitives exploit the properties of the notation described in propositions 32-24 in order to provide services supporting actors in defining, manipulating, and using the C$^2$Ms.

While the distinction between the $\beta$-level and the $\alpha$-level of the notation can be recognized in almost all recent CSCW applications, the $\gamma$-level is characteristic of Ariadne. Moreover, the three levels are not hierarchical, and going from one level to the lower level is not identical to making a refinement, since the information handled at each level is of a different nature. Rather, each level defines the ‘space of possibility’ for the lower one: at the $\alpha$-level one can specify the protocols defined at the $\beta$-level whereas the $\gamma$-level provides the grammars that can be applied at the $\beta$-level to define such protocols. More specifically:
At the $\gamma$-level, it is possible to define or modify the grammar used to construct an infinite variety of protocols at the $\beta$-level. Building a grammar means determining the expressive power of a language for defining a class of C2Ms, that is, the components that will constitute the C2M together with their structural interrelationships (e.g., in order to model workflows according to different approaches, classification schemes, etc.), and define the operational semantics associated (compositionally) to the elements of the grammar. The ‘space of possibility’ within which grammars can be defined at the $\gamma$-level is determined by the set of objects of articulation work, as defined by model of articulation work, and by the set of formal structures for representing various relations (causal, hierarchical, instrumental, etc.) among the objects (the Basic Elements of figure 6).

At the $\beta$-level, it is possible to define or modify the protocol itself according to the selected grammar. For instance, a bug report protocol can be specified with a more or less emphasis on its distributed features, according to the user standpoint, by selecting an appropriate grammar. In this context, the user can determine the allocation of functionality between human actor and protocol, select methods for handling partial specifications, and make permanent changes to an existing protocol as part of its evolutionary design.

At the $\alpha$-level, it is possible to incrementally instantiate and activate the protocol in a particular situation. The same protocol, e.g., the bug form protocol, can be executed repeatedly and concurrently by different actors. Moreover, the primitives at this level allow for the management of local changes.

The current version, of Ariadne incorporates very elemental versioning facilities, both at the $\alpha$- and $\beta$-level. Recent proposals (e.g., Bogia and Kaplan, 1995), suggest more sophisticated facilities for managing evolutionary design which are based on refinements of the representations and on flexible ways of selecting and combining them during the definition and specification of the given protocol.

**Proposition 35.** Ariadne has a three-level structure where each level is specialized to the management of different aspects of a C2M’s definition and specification through appropriate sets of primitives.

The different levels of Ariadne can be accessed by users with different skills. At the $\alpha$- and $\beta$-level, the use of the notation does not require specialized skills, merely the ability of combining and selecting predefined items according to the rules of a relevant grammar and the associated semantics. The $\alpha$- and $\beta$-level are typically needed by end-users who, as part of their everyday work activities, use and design coordination mechanisms. The $\gamma$-level is typically the realm of the ‘application designer’ or, in our framework, actors in charge of defining
grammars needed by end-users to define protocols and of enriching the basic elements for defining the grammars.¹

Referring again to figure 6, the arrows to the left connect the basic elements at the $\gamma$-level with the framework in which they are developed. The connection to the Programming Environment is intended to stress that the notation is in a dynamic but disciplined relationship with its development environment, in the sense that any increase of the expressive power of Ariadne is realized by enriching the sets of basic elements of the notation while otherwise preserving the properties of the notation.

The notation also has an obvious connection with the User Interface Service. As alluded to earlier, the design of Ariadne is concerned with the definition of the information necessary to design C²Ms (that is, the appropriate expressive power of the language) and not with how this information is requested or presented to the users. By this choice we are not denying the crucial role of the ‘material format of the artifact’ (proposition 14) and, more generally, of the whole C²M, as it is represented by the user interface. To the contrary, we are claiming that the design of the (material) representation of the C²Ms requires, per se, a non-trivial and specialized research effort that is beyond the scope of Ariadne. However, Ariadne points to precise requirements for the User Interface Service. Firstly, the various types of elements constituting the notation and how they are composed to construct C²Ms as well as their behavior (the basic elements, their relations, the Interoperability Language) require the user interface designer to look for graphical/multi-media representations which are adequate for presenting C²Ms and their behaviour at the semantic level of articulation work. This representation should exhibit the same properties (compositionality, malleability, linkability) that characterize the notation. Secondly, in accordance with the layered structure of the notation and its interface to the field of work, the presentation should be tailorable to different organizational roles in different application domains. The customization of the presentation according to users’ profiles can employ the techniques derived in the area of User Modeling (Kobsa and Wahlster, 1989) in order to make CSCW systems adaptive (Dimitri and Simone, 1994).

The arrows on the right-hand side of figure 6 connect the C²M with the field of work and the cooperative work arrangement, that is, to its context of use (proposition 21). At the $\gamma$-level there are no connections to the context of use. In fact, the grammars are context-independent in that they simply define the expressive power of the languages potentially used to define specific protocols. By contrast, the definition and specification of a protocol are related to the given field of work and the work arrangement. Firstly, the prospect of using the grammar for defining a particular class of protocols influence the selection of the grammar so as to take into account the nature and complexity of the protocol to be constructed. For example, if concurrency is needed then a grammar providing this

¹ Notice that while the construction of grammars could be performed by end-users with a specific aptitude and competence for modelling, the design of new elements for the notation is a pure programming activity that requires specialized technical skills.
feature has to be selected; if not, it is possible to use a less powerful grammar. Secondly, the grammar determines the relevant objects of articulation work, their relationships and the associated behavior. Thirdly, the objects of the field of work and of the work arrangement are related to the objects of articulation work: as ‘types’ at the $\beta$ level and as ‘instances’ at the $\alpha$ level. Within protocols, ‘types’ are imported together with the related ‘methods’ that are conveyed to the ‘instances’ in the standard way. Thus, Ariadne explicitly requires a clear interface between the C$^2$Ms and their context of use and provides features for defining such an interface.

*Proposition 36.* Ariadne defines the precise interface of a C$^2$M to the field of work and the work arrangement, as represented by the data structures and the functionalities of the application at hand and the wider organizational context.

*Proposition 37.* The expressive power of Ariadne can be increased simply by modifying the set of basic elements at the $\gamma$-level through the interface between this level and Ariadne’s programming environment.

We conclude this section with some comments on the realization of Ariadne. In the initial development of the Ariadne notation a decision was made to deliberately postpone the implementation and concentrate on developing a formal specification of its elements and on evaluating it against the demands derived from field studies. This strategy was adopted, consciously and explicitly, in order to avoid having the notation, implicitly and uncontrollably, influenced by the inevitable limitations of currently available implementation platforms. A ‘concept demonstration’ of the formal specification of the notation has recently been implemented in an environment which is particularly suitable to managing relational structures and their behavior due to software components that can simulate and check properties of the obtained protocols (Simone et al., 1995a). This partial implementation shows that the compositionality of Ariadne together with is layered structure induces a architecture of its implementation that can be mapped in a natural way onto an agent based architecture (Divitini et al., 1995; Simone et al., 1995b) where the Interoperability Language that expresses how C$^2$Ms and their components interact constitutes the agent communication language. We envision a new implementation once a development environment suitable to support this type of architecture has been identified.

8. Computational coordination mechanisms — an approach to CSCW systems design?

We have introduced the concept of malleable and linkable coordination mechanisms as a fundamental device for supporting articulation work and
discussed the basic properties of Ariadne, a notation for the construction of computational coordination mechanisms.

The results achieved so far in the development of Ariadne can be summarized as follows:

1. A methodological approach for the development of an environment targeted to the construction of a specific class of CSCW systems. The development is based on a ‘virtuous circle’ that starts from empirical studies and from the analysis of the features of the existing CSCW applications, reaches the definition of a model of articulation work on which to base the specification of the notation, and finally come back to the field studies in order to assess the notation against their requirements.

2. A systematic way of how to relate the requirements of (a class of) CSCW systems to the properties of the notation for their design (semantic level, compositionality, internal structure, and so forth, as explained in section 6-7).

3. An architecture in which a C²M is defined as an embedded system with a precise interface to the field of work as represented by the application as well as to the cooperative work arrangement.

4. A framework in which a important aspects of the C²M life cycle can be viewed as a distributed activity in which the notation supports the cooperation among the various involved actors. Of course, the notation, per se, does not impose any particular approach to the development of C²Ms; rather, the approach taken depends on organizational choices on how systems are designed, introduced, and maintained in the organization. However, the design of Ariadne was guided by the strong belief that improving and supporting work processes is based on a delicate mesh of top-down and bottom-up changes (Davenport, 1993; Swenson, 1993; Pycock and Sharrock, 1994b; Swenson et al., 1994). The features of Ariadne can be used both ways by defining more or less restrictive policies for the maintenance of C²Ms and for the exploitation of their compositionality. What we want to stress, however, is that the more challenging part of this process, the bottom-up path, is supported by Ariadne through malleability and linkability which, when properly combined, allow for local modification and incremental design of large-scale systems of mutually aligned coordination mechanisms that, as a collective, serve as workflow management systems. On the other hand, because it respects the recursiveness of articulation work, Ariadne allows for the definition of control policies that strengthen or relax the degree of freedom of the bottom-up approach according to a particular view of the wider organization and to the consequent coordination needs.

Now, going beyond Ariadne could take the obvious path of refining and completing its features. The most obvious refinements to be undertaken are the following: (a) enrichment of the articulation work model, e.g., by including time as a basic element in the notation, which will require that a metric of time in the
Coordination mechanisms: An approach to CSCW systems design

articulation of cooperative work is identified and formalized (Goody, 1968; Zerubavel, 1979; Egger and Wagner, 1992; Egger and Wagner, 1993); (b) improvement of the support to the various primitives in terms of capability of governing changes together with their impact and propagation; (c) identification of possible new interoperability modes; (d) further development of Ariadne’s conceptual and computational capability to meet the the demands of a powerful management of evolutionary design and partial specification.

A complementary strategy is to explore the extent to which the concept of coordination mechanisms is able to serve, after all, as the general approach to CSCW systems design. In developing Ariadne, we have focused on providing support for a category of cooperative activities characterized by a high degree of complexity and for which the design of CMs is justified and required. What needs to be explored, however, is whether the notion of malleable and linkable coordination mechanism can be taken as a general foundation for developing any kind of CSCW systems and, consequently, whether the basic features of the notation could be adapted to this more general goal.

We have reasons to think that the answer to the above two questions will be positive, as we are going to explain in the following.

Firstly, the distinction between C2Ms and field of work applies also to the class of systems that are ‘complementary’ to the ones considered in the design of Ariadne: in accord with current usage, we call these systems ‘shared workspaces’. By this term we understand systems that support cooperative work and its articulation through a shared ‘medium’ (either a channel, as in video-conferences, or a set of objects, as in co-authoring systems). However, such systems induce certain — secondary — articulation activities, namely those of articulating the use of the medium itself such as registration, lock, floor control, undo policies, and so forth. While these derived articulation activities to some extent are carried out in an ad hoc manner through the medium itself, there is ample evidence that computational protocols are required to handle the complexity of this special case of articulation work, possibly supported by artifacts (cf., for example, figure 3A in Greenberg, 1991). Secondly, the flexibility of these computational protocols is generally stated as the basic requirement in order to ensure that the system can be adapted to the diverse situations in which cooperation takes place (Greenberg, 1991; Shen and Dewan, 1992; Munson and Dewan, 1994; Choudhary and Dewan, 1995). Moreover, in these frameworks there is the need of a smooth transition across the various protocols governing the articulation of this type of cooperation: this obviously refers to some notion of linkability. And, thirdly, there are attempts to find ‘general models’ for the above features, in order to identify basic elements that can be combined to define and to adapt policies (Greenberg, 1991; Greenberg and Marwood, 1994; Choudhary and Dewan, 1995).

These results can be reconsidered and rephrased in terms of the conceptual framework underlying Ariadne. What is to be changed is the model of articulation work that governs the identification of the basic elements to be composed at the various levels. The relationship between the coordination mechanism and the field
of work remains of the same nature (see figure 6) even if the objects and the processes of the latter might be quite different. Taking the same general approach applied in the development of Ariadne, the required model could be derived from existing proposals and from empirical studies focusing on this specific phenomenon (cf., e.g., Dourish and Bellotti, 1992). In this effort, a crucial point will be to understand the nature of the involved artifacts and the features by means of which they might play an active role in articulation work.

When this has been achieved, one can then, we claim, build a truly comprehensive notation from Ariadne to specify CSCW systems encompassing malleable and linkable shared workspace protocols as well as flexible and linkable coordination mechanisms for the articulation of complexly interdependent distributed activities. This comprehensive notation is expected incorporate the main features of a CSCW platform conceived of as an extension of existing operating systems. This CSCW platform will be able to support different CSCW facilities and applications which can co-exist without redundancy of information about their field of work and the work arrangement and which can interoperate through their respective coordination mechanisms.

We can make the above scenario more palpable by illustrating (figure 7) the relation between the comprehensive notation (Ariadne*) and one of the recent platforms proposed for the design of CSCW applications (Trevor et al., 1994). Due to the demanding properties of the notation, the platform constituting its programming environment must possess at least two basic capabilities: (a) an object-oriented view that in a natural way implements compositionality and (b) features to implement notifications at the user level as well as at the level of the objects themselves. The above platform has been designed by specifically addressing such requirements. The capability of managing notifications is fundamental since each component, and by consequence each C2M, is designed so as to be able to convey information about the changes of its internal states to its context as well as to the user interface.
The diagram shows how the Ariadne notation is conceived of as an extension to the considered platform taken as an example of an advanced platform especially oriented to support shared workspaces and their visualization. This extension complements the capabilities of the platform by providing the designer with specific features tailored to the design of coordination mechanisms for the articulation of complex distributed activities. The resulting platform then, eventually, makes it possible to effectively integrate CSCW applications requiring any type of support to articulation work.

Acknowledgments

The development of the framework outlined in this paper is supported by ESPRIT Basic Research through Action 6225 (COMIC or Computer-based Coordination mechanisms in Cooperative Work) and by the Danish Research Council for the Natural Sciences. The framework has been developed in collaboration with our colleagues in the COMIC project, especially our colleagues at Risø and Milano: Hans Andersen, Peter Carstensen, Monica Divitini, Betty Hewitt, Alberto Pozzoli, and Carsten Sørensen. We are especially indebted to Peter Carstensen and Bjarne Kaavé for generously sharing their field study findings with us and to Liam Bannon for shelling the manuscript with a barrage of comments.
References


Divitini, Monica, Carla Simone, Kjeld Schmidt, and Peter Carstensen (1995): A multi-agent approach to the design of coordination mechanisms. WPCS-95-5, Roskilde University, DK-4000 Roskilde, Denmark.


Schmidt, Kjeld, Carla Simone, Monica Divitini, Peter Carstensen, and Carsten Sørensen (1995): A ‘contrat sociale’ for CSCW systems: Supporting interoperability of computational coordination mechanismsWPCS-95-7, Roskilde University, DK-4000 Roskilde, Denmark.


Chapter 2

Coordination Mechanisms
in a multi-agent perspective

Monica Divitini, Carla Simone, Carla Quaranta Vogliotti
DSI - University of Milano

1. Introduction

The previous chapter illustrated the main motivations leading to the notion of Coordination Mechanism (CM) as a means to govern the complexity of articulation work. Moreover, CMs have been defined in terms of their constitutive components, namely an artefact together with the concomitant procedures and conventions governing its use for coordination purposes. The aim of constructing computer based supports to articulation work led to the definition of 'computational' CM (C2M) which is based on the notion of CM in that it contains exactly the same components, and extends the CM capability by exploiting the possibility of allocating functionality from the actors to the computer artefact. This allocation of functionality has to be realized without imposing undue rigidity to the articulation work support. The investigation about how CMs are used in real work settings led to the identification of two basic requirements, namely malleability and linkability (as explained in chapter 1).

Now the construction of C2Ms that meet these requirements has to be based on a notation suitable to provide adequate primitives to adapt C2Ms to the varying needs of their context of use as well as to combine more elemental C2Ms into a C2M supporting articulation work by taking into account a broader organizational context.

The scope of the present chapter is to describe the main features of Ariadne, a notation for the construction of malleable and linkable C2Ms. The rationale behind Ariadne has been discussed in the previous chapter (§ 1.6). Here we illustrate how this rationale leads to a language formally defined both in its syntax and operational semantics. In doing this, we will follow the natural path connected with the structure of Ariadne layered in three levels (see figure 2.1.1, derived from chapter 1), by starting from the $\gamma$-level down to the $\alpha$-level.
This chapter can be ideally divided into two parts. The first part (section 2.2) defines the elements of Ariadne; the second part (section 2.3 and section 2.4) propose the notion of 'agent' as a means for defining a high level implementation of Ariadne. The basic idea is to realize each element of Ariadne as an agent and to define how agents are combined to construct a C2M in terms of the definition of each agent's communication capabilities according to the constituents of the Interoperability Language. Section 2.5 shows an example of construction of a C2M. Section 2.6 discusses how the primitives of Ariadne can be interpreted in the above multi-agent architecture of a C2M. This second part can be viewed as the specification of the software architecture of the demonstrator of Ariadne that is described in Chapter 9.

2. The elements of the notation

As argued in the previous chapter, one of the main properties of Ariadne is compositionality, meaning by that the possibility of composing its elements in a flexible way to obtain the desired expressive power. In order to achieve this goal, each element of Ariadne has to be described in terms of its relations to the other elements. The standpoint taken in this section is the one of the articulation work: then, the description of the relations is given in a 'qualitative' way, both in terms of the syntax and semantics. In other terms, here we are interested in the type of information we want to get from who wants to construct a C2M, not in how this information is translated in the internal representation of the C2M; this is the matter of the next sections. The following description can be therefore seen as a
sort of rudimentary user interface for data acquisition. This section is a revision of some parts of Chapter 3 of Deliverable 3.3 (Simone et al., 1994). The aspects not handled here, namely the Formal Structures and the primitives at the β- and α-level, have to be considered as defined in the above mentioned Deliverable.

### 2.2.1. A description of the OAWs

In this section we describe the Objects of Articulation Work (hereafter OAW) as a list of attributes with the related type. The attributes of the OAWs are chosen on the basis of the operations on the objects that are described in the articulation work model (see chapter 1 - section 6) How the attributes specify the behaviour of the OAWs will be explained in section 2.4.1, after the introduction of the Interoperability Language.

In the description of the OAWs we shall adopt the following conventions: X* denotes a finite sequence of items of sort X; Condition is a boolean condition and out-trigger is a communicative event (the exact nature of this two types will be explained in section 2.4). Moreover, a policy is a set of rules or the invocation of another C2M.

**OAW :: = Role/ Actor/ Task/ Activity/ Action/ Resources**

where

**Role =**

<table>
<thead>
<tr>
<th>ATTRIBUTE NAME</th>
<th>ATTRIBUTE TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>data-frame</td>
</tr>
<tr>
<td>responsible for</td>
<td>Resource*</td>
</tr>
<tr>
<td>responsible for</td>
<td>Task *</td>
</tr>
<tr>
<td>responsible for</td>
<td>C2M*</td>
</tr>
<tr>
<td>involved in</td>
<td>C2M*</td>
</tr>
<tr>
<td>precepts</td>
<td>set of rules</td>
</tr>
<tr>
<td>assumed by</td>
<td>Actor*</td>
</tr>
<tr>
<td>defined by</td>
<td>&lt;Role, policy&gt;</td>
</tr>
<tr>
<td>adapted by</td>
<td>&lt;Role, policy&gt;</td>
</tr>
<tr>
<td>awareness</td>
<td>&lt;Condition, out-trigger&gt;</td>
</tr>
</tbody>
</table>

A role is defined through a set of responsibilities for Tasks, Resources and C2Ms. These reponsibilities are established in the Organisational Context by the Role mentioned in the attribute **defined by**. The definition of a Role can be changed by some enabled Role, still in the Organisational Context, namely by the role mentioned in the attribute **adapted by**.

A role can be **assumed by** one or more actors. The actor, after assuming a role, becomes a committed actor: this event marks the passing from nominal to actual

---

1 See Chapter 1- section 6 for a discussion of the requirements of Ariadne's User Interface.
status of the C2M. The rules in the attribute **precepts** regulate the assumption of the role by an actor and the behaviour of the actor that assumes the role. For example, a role can be assumed by people carrying a certain experience or possessing some formal attributes (like, a PhD); the role has to refer, by default, to another role for handling exceptional situations in the tasks it is responsible for.

The attributes **defined by** and **adapted by**, as stated above, describe the possibility of the definition and modification of the Role by some enabled Role belonging to the Organisational Context.

The attribute **awareness** that appears in the definition of the role expresses the capability of the object to make other OAWs aware both of its internal state and of communications received by its environment.

These last three attributes are owned by all the OAWs and have the same purposes: therefore we shall not mention them in the explanation of the other OAWs.

**Actor**

<table>
<thead>
<tr>
<th>ATTRIBUTE NAME</th>
<th>ATTRIBUTE TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personal Data</td>
<td>data-frame</td>
</tr>
<tr>
<td>locality</td>
<td>place</td>
</tr>
<tr>
<td>skill</td>
<td>description</td>
</tr>
<tr>
<td>assigned to</td>
<td>Role*</td>
</tr>
<tr>
<td>committed to</td>
<td>Task*</td>
</tr>
<tr>
<td>initiator of</td>
<td>Activity*</td>
</tr>
<tr>
<td>doer</td>
<td>Action</td>
</tr>
<tr>
<td>assigned by</td>
<td>&lt;Role, Policy&gt;</td>
</tr>
<tr>
<td>awareness</td>
<td>&lt;Condition, out-trigger&gt;*</td>
</tr>
</tbody>
</table>

An actor is a person characterised by her/his **personal data**, her/his **skills**, the **locality** where s/he lives in the organization, the set of Roles s/he assumes (attribute **assigned to**) and the Tasks s/he is **committed to**. These latter are a part of the Tasks which the Roles that the Actor assumes are responsible for. Furthermore, to accomplish a Task, an actor has to start activities and do actions. The action that an actor is doing is not necessarily part of one of the activities s/he started; an actor, in fact can start an activity and delegate other actors to do the actions that are part of the activity. Finally, the Role in the attribute **assigned by** is the one in the Organisational Context which assigns the Actor to the Roles it assumes. The assignment is governed by the rules stipulated by the attribute **precepts** of the Role to be assigned: for example, the precepts discussed above. Moreover, it is possible to specify a **policy** to be followed when the role is assigned. For example, the policy could state that the assignment has to be approved by some other role, possibly via the activation of a specific C2M.
A task is characterised by its description, preconditions, priority and postconditions. In-triggers are events that must be listen to before the task can start. The responsible role is the one that activates and accomplishes the task, whereas the role in the attribute supervised by must be contacted in case of exceptions. Referring to the nominal part of the articulation work, a Task can potentially make use of a set of resources. These latter can also be human resources, that is a set of person that can do the activities realising the task. This set of potentially used resources is described in the attribute has allocated. As a distributed entity, a task is potentially realised by a set of activities: a nominal task becomes an actual activity, chosen in the set described in the attribute realised by. Criteria of accomplishment is a policy that the responsible role follows to decide the termination of the task. The roles mentioned in the attributes assessed by and approved by are in charge of the declaration that the task has been accomplished in a satisfactory way: these roles can be expressed both in an absolute way or defined by a relation with the role responsible for the task. The role in the attribute assigned by belongs to the Organisational Context: it assigns the responsibility of the task to a role.
An **Activity** is a structure of **Actions**; where a structure is one of the relational structures introduced in § 3.3.1 of (Simone and Schmidt, 1994). These relations express the causal relation binding the actions constituting the activity.

An action can be a **PerfAction** or an **Interaction**.

**PerfAction** =

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Attribute Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>identifier</td>
</tr>
<tr>
<td>Content</td>
<td>description/FowProcedure</td>
</tr>
<tr>
<td>Realises</td>
<td>Task</td>
</tr>
<tr>
<td>initiated by</td>
<td>Actor*</td>
</tr>
<tr>
<td>done by</td>
<td>Actor*</td>
</tr>
<tr>
<td>in, out, used</td>
<td>Resource*</td>
</tr>
<tr>
<td>in-triggers</td>
<td>In-message</td>
</tr>
<tr>
<td>precondition</td>
<td>test</td>
</tr>
<tr>
<td>priority</td>
<td>priority value</td>
</tr>
<tr>
<td>postcondition</td>
<td>test</td>
</tr>
<tr>
<td>state</td>
<td>state-value</td>
</tr>
<tr>
<td>temporal constraints</td>
<td>deadline</td>
</tr>
<tr>
<td>duration</td>
<td>time-interval</td>
</tr>
<tr>
<td>awareness</td>
<td>&lt;Condition, out-trigger&gt;*</td>
</tr>
</tbody>
</table>

where state-value belongs to the set: \{waiting, ready, started, under-execution, concluded, error\}.

A performative action can be described as a call to a procedure of the field of work or as a computation (the attribute **content**); it has **priority**, **preconditions** and **postconditions**; furthermore an action, as a part of an activity, **realises** a task.

As explained above, an activity can be initiated by an actor and then its actions can be delegated to other ones. The actor in the attribute **initiated by** is the one who started the activity which the action is a part of, whereas the actor in the attribute **done by** is the doer of the action, which can be different from the previous one.

To get the action done, some resources are needed: they are expressed in the attribute **in, out, used**, as they can be input, output resources or resources that are necessary. These attributes are the actual counter-part of the attribute **has allocated** of the Task: some of the resources that potentially a task can need are actually allocated to the activity (thus to the various actions) realising the Task.

The **in-triggers** have the same meaning as in the case of a task. Finally, an action has a **temporal** attribute to express **constraints** and deadlines and a **state** expressing its degree of achievement. The state error can be checked to activate
exception handling processes that typically involve the supervisor of the task the action contributes to realize.

\[ \text{InterAction} = \]

<table>
<thead>
<tr>
<th>ATTRIBUTE NAME</th>
<th>ATTRIBUTE TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belongs to/Realises</td>
<td>Proctor/Task</td>
</tr>
<tr>
<td>sender</td>
<td>Role(Actor)^*/AA</td>
</tr>
<tr>
<td>content</td>
<td>InfoRes*</td>
</tr>
<tr>
<td>type</td>
<td>&lt;must, may&gt;</td>
</tr>
<tr>
<td>receiver</td>
<td>Role(Actor)^*/AA</td>
</tr>
<tr>
<td>ip</td>
<td>Illocutionary Point</td>
</tr>
<tr>
<td>precondition</td>
<td>test</td>
</tr>
<tr>
<td>answer time</td>
<td>deadline</td>
</tr>
<tr>
<td>awareness</td>
<td>&lt;Condition, out-trigger&gt;</td>
</tr>
</tbody>
</table>

Since the notation describes a distributed model of cooperative work, exchanges of information among the entities constituting a C2M are necessary. It should be distinguished between two kinds of interaction: in the nominal definition an interaction can be a part of a proctor (see below), that is, it represents an exchange of information among Roles and/or the AA. When the C2M is in the actual status, an interaction can be a step of an activity and in this case it realises a task.

Then, we should distinguish among the nominal and the actual definition of an interaction. In the nominal definition the sender(s) and the receiver(s) are Roles or the Active Artefact, that is the component of the notation capturing the notion of the symbolic artefact of a CM (it will be described in the next section). In the actual definition, the sender(s) and the receive(s) are Actors or the AA.

The other attributes are self-explanatory: there must be some content; an interaction has precondition, answer-time and an illocutionary point (Searle, 1975; Winograd and Flores, 1986).

The attribute type indicates if a response to the Interaction from the receiver is mandatory (must) or voluntary (may).
<table>
<thead>
<tr>
<th>ATTRIBUTE NAME</th>
<th>ATTRIBUTE TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>identifier</td>
</tr>
<tr>
<td>Type</td>
<td>InfoRes-type</td>
</tr>
<tr>
<td>Description</td>
<td>Text</td>
</tr>
<tr>
<td>Managed by</td>
<td>Role</td>
</tr>
<tr>
<td>Allocated to</td>
<td>Task</td>
</tr>
<tr>
<td>Access rights</td>
<td>&lt;Access-type, policy&gt;</td>
</tr>
<tr>
<td>Location</td>
<td>Space</td>
</tr>
<tr>
<td>Relation</td>
<td>&lt;Resource-name, rel-type&gt;*</td>
</tr>
<tr>
<td>State</td>
<td>state-value</td>
</tr>
<tr>
<td>Governed by</td>
<td>&lt;Role, policy&gt;</td>
</tr>
<tr>
<td>Defined by</td>
<td>&lt;Role, policy&gt;</td>
</tr>
<tr>
<td>Adapted by</td>
<td>&lt;Role, policy&gt;</td>
</tr>
<tr>
<td>awareness</td>
<td>&lt;Condition, out-trigger&gt;</td>
</tr>
</tbody>
</table>

where:

InfoRes-type belongs to the set: \{documents, letters, applications, notes, files, memos, reports, drawings\}
access-type belongs to the set: \{read, write, copy, move, transfer, load, unload\}
rel-type belongs to the set: \{definition, classification, prototypical, causal, genetic, historical, means-end, part/whole relation, copy of\}
state-value belongs to the set \{allocated, in-use, in-use-by\}

The attribute Relation indicates some relation between the InfoRes and other resources (e.g. an InfoRes can be a copy of another one)

The Role responsible for the resource, when it allocates a Resource to a Task, must follow the access rights with the related policies.

The attribute Managed by is referred to the Organisational Context (nominal status), while the attribute Governed by is referred to the OAW level (actual status): the role in this attribute is the role responsible for the resource, whereas the role in Managed by is the one which assigns the responsibility of the resource to the responsible role.

All the other resources have the same structure: they only have different types of attributes or different sets of values, according to the Articulation work model:

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 are determined by the field of work.

2.2.2. The Active Artefact and the Proctors

A CM has been defined as a protocol consisting of a set of prescribed procedures and supported by a symbolic artefact; thus to define a C2M one must specify the Active Artefact (AA), capturing the notion of symbolic artefact and some structures capturing the notion of prescribed procedures.

The Active Artefact is described using the same conventions used for OAWs. The introduced attributes are instrumental to the representation of the global state of the C2M and of the links between this latter and the FoW. Moreover, some attributes describe AA’s ability to notify to its environment appropriate information in presence of particular conditions. In particular, since the Active Artefact actively participates to the articulation work effort, it is appropriate to associate to it coordination capabilities.

ActiveArtefact =

<table>
<thead>
<tr>
<th>ATTRIBUTE NAME</th>
<th>ATTRIBUTE TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>identifier</td>
</tr>
<tr>
<td>visibility</td>
<td>&lt;Role, type&gt;*</td>
</tr>
<tr>
<td>content</td>
<td>data-frame</td>
</tr>
<tr>
<td>update/read requests</td>
<td>&lt;Role, request&gt;*</td>
</tr>
<tr>
<td>coordination</td>
<td>&lt;Condition, out-trigger&gt;*</td>
</tr>
<tr>
<td></td>
<td>&lt;In-trigger, function&gt;*</td>
</tr>
<tr>
<td>awareness</td>
<td>&lt;Condition, out-trigger&gt;*</td>
</tr>
</tbody>
</table>

In the attribute visibility, type can assume the value <update, read>; this attribute gives an account of the way each Role in the C2M can access the information contained in the AA. This information is stored in the attribute content whose type (data-frame) is a sequence of pairs <slot-name: slot-type>. The update/read requests are the communication the AA is prepared to receive: a request can be a read or an update, which has to be consistent with the type specified in the attribute visibility. Coordination and awareness express AA’s capability to propagate, among the C2M’s components, information about changes to the state of the C2M induced by some of them. The condition triggering coordination and awareness can be based both on AA’s internal states and on some communication event plus some constraints on the content of the received message: when the condition is satisfied, the AA sends a message which the receiver(s) must (coordination) or may (awareness) listen to. Accordingly, the message can be broadcasted or sent to a destination explicitly specified in the out-
triggers. Furthermore, the coordination can consist of the activation of some internal function as a consequence of the receiving of some messages which meet some conditions.

Let us stress that coordination is a peculiar attribute of the AA; no OAW has communication of the type 'coordination'. The motivation relies on the fact that OAWs do not have, per se, an autonomous behaviour, rather they are just contributing to the definition of the overall C2M. Then it is meaningless to coordinate them with other C2M behaviour. Instead, they are able to monitor their modifications and notify them as expressed by attribute awareness. In other words, OAWs possess a little degree of reactivity.

A proctor is a structure describing a procedure, at the nominal level. A proctor is a compound entity obtained from the composition of OAWs by means of formal structures. Actually, a procedure in a cooperative arrangement can be expressed in terms of relations among Tasks, Roles, Actors, Performative Actions, Interactions and Resources.

<table>
<thead>
<tr>
<th>ATTRIBUTE NAME</th>
<th>ATTRIBUTE TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>identifier</td>
</tr>
<tr>
<td>Description</td>
<td>Partial order relation $\subseteq$ OAWs $\times$ OAWs</td>
</tr>
</tbody>
</table>

In accordance with the claim made in chapter 1, Ariadne does not propose a unique approach to the representation of procedures. Then, the selection of the OAWs and of their relations is defined by the grammar the designer of the C2M wants to adopt for its definition.

For example, one could select a graph as support of the relational structure and interpret its nodes as Roles and label the arcs as Interactions. These latter specify the type of resources flowing from a role to the related one in the graph. Another typical way of representing procedures is based on formal structure, like Petri nets, which represent a causal relation among nodes interpreted as job to be done. This modeling approach will be taken in the example illustrating the notation (see section 2.5) where a procedure is described as a set of Tasks to be accomplished plus some prescribed Interactions among the Roles responsible for the Tasks, and the AA. Then, in this case proctors are described by means of a causal relation $\subseteq (\text{Tasks} \cup \text{Interactions}) \times (\text{Tasks} \cup \text{Interactions})$.

Proctors definition is the first example of application of the modularity characterizing Ariadne. This is reflected also at the level of their behaviour, as we shall describe in the next sections. In fact, the behaviour of a proctor is obtained from the composition of the behaviour of the Tasks, the Interaction the partial order relation appearing in its definition.
A C2M consists of a collection of proctors and the related Active Artefact. It is then described as follows:

\[
\text{C2M} = \begin{array}{|c|c|}
\hline
\text{ATTRIBUTE NAME} & \text{ATTRIBUTE TYPE} \\
\hline
\text{name} & \text{identifier} \\
\text{Active Artefact} & \text{AA} \\
\text{Procedures} & \text{Proctor} * \\
\hline
\end{array}
\]

As widely discussed in strand1, real working situation require a flexible combination of C2Ms to support articulation work. Then linkability of C2Ms is one basic requirement that the notation has to satisfy.

As argued in chapter 1, the empirical studies allow one to identify three kinds of linking modes among CMs: subscription, inscription and prescription modes.

In order to support the definition of the links among C2Ms Ariadne exploits the notion of interface as a means to manage these links and to facilitate the construction of composed C2M (Genesereth and Ketchpel, 1994).

Each C2M has associated an interface which is in charge of managing the interaction among various C2Ms. In particular an Interface must specify the access rights to the AA and the possible interactions with other C2Ms with respect to the three linking modes. These modes are specialised into different attributes which describe the form taken by each linking mode and the component involved by the linking mode.

\[
\text{C2M-Interface} = \begin{array}{|c|c|}
\hline
\text{ATTRIBUTE NAME} & \text{ATTRIBUTE TYPE} \\
\hline
\text{name} & \text{identifier} \\
\text{External Visibility} & <\text{C2M, AA}-\text{attribute, type}> * \\
\text{Subscription} & <\text{Role, C2M, format}> * \\
\text{Coordination} & <\text{C2M, in}/\text{out, ack}> * \\
\text{Awareness} & <\text{C2M, in}/\text{out, ack}> * \\
\text{Control} & <\text{C2M, Action/Task}, \text{pre}-\text{conditions}, \text{post}-\text{conditions}> \\
\text{Prescription} & <\text{C2M, component, in}/\text{out, nominal/actual}> \\
\hline
\end{array}
\]

The attribute \text{External Visibility} states the type (read/update) of visibility of the current C2M’s AA from the linked C2Ms: the visibility type can be different for each part (attribute) of the data frame describing the AA.

The \text{Subscription} attribute describes the possibility for the C2M to interoperate with other C2Ms in the subscription mode: in particular a Proctor
belonging to the current C2M can make the behavior of another C2M part of its behavior in the sense that it activates it or it is called to participate to it.

The **Prescription** attribute expresses the possibility of a C2M to be modified in one of its components by another C2M (in), or to modify (out) the definition of components of another C2M. This modification can affect the nominal definition or actual specification of the component.

The **Coordination** and **Awareness** attributes describe the exchange of information with other C2Ms in the inscription mode; these attributes specify with which C2M the exchange is possible and in which direction (that is if the current C2M is allowed to receive or send information). In addition the value of ack, compulsory for coordination and voluntary for awareness, fixes the kind of reaction of the target C2M.

The **Control** attribute expresses a synchronisation among the execution of Proctors belonging to different C2Ms. For example, some Tasks of the current C2Ms must be synchronously executed with Tasks belonging to other C2Ms. This case is can be viewed as a generalization of the inscription mode when the pattern of interaction is not fully specified but for the verification of some pre- or post-conditions governing the start and the termination of the synchronization. An example of generalized inscription mode is given in strand1. Figure 2.2.1 describes how the Bug Handling coordination mechanism (BH), introduced in chapter 1, is linked to other coordination mechanisms in terms of the linking modes.
3. The agent perspective

Ariadne allows actors to construct a C2M by means of the elements of the notation described in the previous section. What has to be described now is the operational semantics associated to each of these elements, in order to make the description a computational support to articulation work. To this aim, we exploit the notion of agent that fits well with the main characteristic of Ariadne, namely compositionality through well established interfaces among elements.

First of all, the notion of agent is discussed. Then, the elements constituting Ariadne are organized in a multi-layer agent architecture. Further, the agent communication language, called Interoperability Language, is presented, and finally used to describe the agents' communication capabilities.

Is is worthwhile to recall that the elements of Ariadne have been identified on the basis of empirical studies, as illustrated in the Chapter 1. The same holds for the Interoperability Language. One of the main point in this section is to show that
this language can be uniformly used in the architecture not only for formalizing the information provided by the designer of C2Ms but also to 'implement' the implicit (that is, system-defined) interactions among all the elements of the notation. In this way, the operational semantics of Ariadne is given in a formal way that, in addition, defines in a straight way the software architecture of its implementation, as it will be described in Chapter 9.

2.3.1. The notion of agent

When we try to give an answer to the question 'What is an agent?' it is necessary to face the fact that the term is widely used by many people working in closely related areas, making difficult to produce a single universally accepted definition.

The term 'agent' was used at the beginning by Minsky in his "Society of Mind", where he treats agents as individually very simple, but giving rise to "intelligence" when acting together, in certain very special ways, in societies" (Minsky, 1985).

At the moment, when people speak about agent there are at least three main interpretation. Agents can be seen as:

(i) mentalistic entities: In (Shoham, 1993), agents are seen as having beliefs, commitments, capabilities, etc., without any explicit implementation of "intelligence".
(ii) entities that communicate via a common language (Genesereth and Ketchpel, 1994)
(iii) something/one that acts on your behalf

In order to answer the question 'what is an agent?', Wooldridge and Jennings (Wooldridge and Jennings, 1995) distinguish between a weak and a strong notion of agency.

Following the weak notion of agency, "...the term agent is used to denote a hardware or software-based computer system that enjoys the following properties:

• autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their internal state;
• social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;
• reactivity: agents perceive their environment and respond in a timely fashion to changes that occur in it;
• pro-activeness: "agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative... " (from (Wooldridge and Jennings, 1995) ).

This notion of agency buys us a useful computational metaphor and abstraction tool. An agent is, in a certain way, a kind of Unix-like software daemon that can receive task to accomplish, cooperate with other agents in order to achieve these tasks and take the initiative, but only in a limited way.
This is probably the notion of agency more widely used outside the AI community. This latter exploits a stronger notion according to which an agent is a computer system that has the previous properties and is moreover conceptualized using concepts usually applied to people, like mentalistic notions and rationality. In other words, AI defines agents as intentional systems. This term was coined by the philosopher Daniel Dennett to describe entities "... whose behaviour can be predicted by the method of attributing belief, desires and rational acumen..." (Dennett, 1987).

To the different ways of seeing agents correspond to different agent languages, i.e. different computer system which allows one to develop and program agent-based computer systems. Here too Wooldridge and Jennings distinguish between two notions of agent languages, the weak ones, used in the software agent community, and the strong ones, typical of AI.

As an example, in the first group we find Hewitt & Agha's ACTOR languages (Agha, 1986), TCL/TK and TELESCRIPT; in the second one we find AGENT0/PLACA and Concurrent METATEM.

A language that does not fit in this classification is KQML/KIF (Genesereth and Ketchpel, 1994; Labrou and Finin, 1994). The interest for this language is growing as it is a step towards the need to develop an approach that separates the issue of language in which an agent is written and the protocol of interaction between agents. KQML (Knowledge Query Manipulation Language) is a message format and a message-handling protocol to support run-time knowledge sharing and interaction among agents. KQML is intended to be, as stated in (Labrou and Finin, 1994), a universal interaction language that supports communication through explicit linguistic actions and that can be used in any environment where software agents need to communicate something more than pre-defined statements of facts.

Although KQML is used in a number of projects, it still lacks a formal semantics. In (Labrou and Finin, 1994), a first attempt has been made in order to define a unambiguous semantics of the language, but much work remains to be done. Though KQML has good probability to become a widely-accepted standard, only a more precise definition and semantics could made agent designers more confident about its use. Going in this direction, Cohen and Levesque identified some of the difficulties with the language, as ambiguity and missing performatives, and presented an attempt to overcome them (Cohen and Levesque, 1995).

From our perspective, the concept of agent, in its weaker notion, has proved to be a very useful abstraction that allows to think of Ariadne as a system constituted by different interacting agents organized at different levels of complexity, as described in the following. Since no existing communication language shows the needed characteristics, it has been necessary to define an Interoperability
Language. A possible mapping of this language in KQML for its implementation is at the moment under study.

2.3.2 A multi-agent multi-level architecture for Ariadne

Following the idea of realizing as an agent each element of Ariadne, and then each component of a C2M, a C2M will be constituted by a collection of reactive agents. Since this applies for any type of C2M, Ariadne can be considered as a notation for defining a multi-agent architecture for supporting articulation work among cooperating actors. In addition, the architecture shows a multi-layer structure since C2Ms are compound entities that are built on top of the basic elements.

The OAWs are the building blocks of C2Ms made available by Ariadne and agents corresponding to them populate the first layer of the architecture. According to the previous section, they are described by means of attributes that can be classified as follows:

a) attributes identifying the object and describing its informational content: they vary from an object to another.

b) attributes representing the relationships with the Organisational Context where OAW are defined and adapted:

c) the awareness attribute expressing the capability to notify information about changes of internal states.

Then, OAW can be modeled as agents specialised to the management of the information characterising them (a), and of the relationships with their environment (b and c). The communicative behavior of the OAW agents will be explained in section 2.4.1.

The second layer, related to each single C2M, is populated by the agents capturing the coordination capabilities of artefacts and procedures.

The simplest case is when a C2M is constituted by an Active Artefact and a single procedure. In this case, the architecture of the C2M contains two agents: the agent modelling the Active Artefact (AA)\(^1\), that is the part of the C2M capturing the structure and the behaviour of the ‘symbolic artefact’ mentioned in the definition of CM and an agent capturing the behaviour of the procedure, that will be called Proctor agent.

In the general case a C2M can contain several Proctor agents. This possibility allows one both to capture the distributed nature of articulation work and to increase the modularity of the overall architecture. Figure 2.3.1 sketches the

\(^1\) For the sake of simplicity, agents will be denoted by the name of the corresponding concepts.
layout of a C2M where the arrows represent Interactions both between Proctor agents and the AA, and between any pairs of Proctor agents.

Figure 2.3.1: The agents constituting a C2M agent.

A complete description of the UI agents is out of the scope of our work. Here, it is sufficient to see them as agents responsible for communicating with the users (in the language of the chosen UI technology) and with the agents constituting C2Ms to convey to them the user’s decisions and commands. In the opposite direction, the UI agent is the destination of the messages concerning user’s awareness of what is going on inside C2Ms and through them in the FoW and Cooperative Arrangement. In this way, our UI agents can naturally incorporate some standard services as filtering (Sheth and Maes, 1993) intelligent assistance (Greif, 1994) based on appropriate User Models (Kobsa and Wahlster, 1989) and on user’s preferences.

The behavior of a C2M is the parallel composition of the behaviors of the related AA and Proctor agents on the basis of their communication capabilities.

Finally, the third layer is populated by the agents obtained from the linking of already existing C2Ms. Specifically, the definition of Proctor agents can subscribe to other C2M agents in order to support a negotiation or in order to activate a process that is coordinated by another C2M. An example of subscription to a C2M is when Proctor agents make reference to a C2M supporting conversations (Winograd and Flores, 1986) among Roles. This C2M is basically constituted by Interactions combined by causal relations and by an AA describing the status and history of the conversation. Through the reference to these types of C2Ms, the communication capabilities of Ariadne's agents (captured by the Interoperability Language, see next section) is therefore very expressive and flexible in terms of patterns of Interactions (Labrou and Finin, 1994).
The linking of C2Ms is defined by specifying the interface each of them presents to the linked C2Ms. The interface establishes which information can be mutually accessed and communicated by the agents constituting the involved mechanisms. Then, as depicted in Figure 2.3.2, the links between any two C2Ms can be in a direction or in both directions.

![Diagram of C2M interfaces](image)

Figure 2.3.2 - links between C2M.

The interface is a sort of wrapper (Genesereth and Ketchpel, 1994) that plays the role of both facilitator and monitor of the embedded C2M in order to handle the additional communication needs derived from the linking of different C2Ms. The interface is again an agent specialized to managing external communication, and, in a more comprehensive view, specialized to support a first degree of tolerance of the modifications implied by the requirement of malleability.

### 2.3.3 The Interoperability Language

In this section we describe the Interoperability Language (IL) that is used by the agents to interoperate with other agents that can also belong to others C2Ms. The three linking modes determine the basic primitives of the language.

The IL contains primitive that take the following general forms

- `<Receiver>![parameter]<message><ack>`
- `<Sender>?[parameter]<message><ack>`

with the following meaning, respectively: the agent in whose specification the communication occurs sends to (!)/receives from (?) the <Receiver/Sender> the content <message>; <ack> is a two-valued parameter: the default value ‘must’ specifies that the reaction by the target agent has to be compulsory, that is, it is required an acknowledgment from the receiver. The alternative value ‘may’ states that the reaction by the target agent is voluntary: it is not needed any
acknowledgment from the receiver, who can use the received information or ignore it. [parameter] can contain a value or be empty: it is specified only in the actual status of the C2M to specify which instances, among those associated to the Sender/Receiver, is involved in the communication.

Each <message> takes one of the following formats, where the operator \(<X>*\) denotes a finite sequence of items of sort X:

\[tell(<variable:value>*):\] the sender wants to communicate to the receiver the values of some variables;

\[ask(<variable>*):\] the sender wants to get information into some variables

\[perform(<command>, <parameter>*):\] the sender wants the receiver to perform the command with the given parameters;

\(<command>\) can be any of the following:

\[<update> with parameters <variable:value>*\], to assign values to variables

\[<read> with parameters <variable>*\], to read values into variables;

\[<activate>, without parameters, to activate the receiver agent; in particular the receiver can be a Task, an Action or a C2M. In the case of partial specification, the activated agent requires to the sender the parameters to start its operations.

\[<participate> with parameter <Role>; to involve the receiver in a C2M; the Role in the parameter is the one who activated the C2M.

\[synch(<CM.Proctor.Action/Task>*), <pre-conditions, post-conditions>\): each Action/Task belongs to a different CM; this message expresses that the Actions/Tasks have to be executed respecting pre- and post-conditions.

\[over-write(<item>*), <new-item>*\): the sender enforces modifications on some items in the receivers. When the first group of parameters takes the value ‘empty’, the command corresponds to the creation of the second group of items.

Both in the case of a \(perform\)(read,...) and an \(ask\)(...), the sender wants to get the value of the specified variables. The main difference is that in the first case the sender knows about the structure of the data owned by the receiver and therefore the command must terminate successfully; in the second case, the variables denote an information structure owned by the sender where the command deposits the asked values, if and when it terminates successfully. An analogous distinction characterises the differences between \(perform\)(update,...) and \(tell\)(...).

The semantics of the IL primitives is defined in the same vein as the communication primitives in Hoare's CSP (Hoare, 1985), from which we also drew our inspiration to define their format. Basically, each primitive carrying a \(!\) has to be executed synchronously with the corresponding primitive carrying a \(?\), and vice-versa, the correspondence being defined as the identity of the involved parameters. This choice is not restrictive since it is well known that asynchronous communication can be modeled by means of a pattern of synchronous communications with a process that plays the role of communication buffer. The
modeling of this situation is out of our scope, and could be added to our framework by applying the techniques proposed, for example, by the Actor Model (Agha and Hewitt, 1987), under the same fairness hypothesis about message handling.

The primitives of the IL are used to describe the behaviour of the various components of the notation. In the next section we shall characterise the operations of the components of the notations at the $\gamma$-level in terms of the interoperability capabilities associated to the attributes carrying the operational behaviour. Then, compositionally, the operational semantics of the whole objects is built up using these ‘pieces of behaviour’, and the behaviour of compound agents is built up using the semantics of each component.

4. The communicative behaviour of the various agents

The aim of this section is to exploit the IL to the definition of the communicative behavior of the agents populating the three levels illustrated in section 2.3.2. The presentation of these behaviors is organized according the three levels of the agent-based architecture.

4.1. The behavior of the Objects of Articulation Work

The behavior of the OAW is based on the behavior associated to their attributes: some of them are common to all OAW and others are specific to each OAW. We start from the former attributes.

As described in section 2.2.1 each OAW is characterized by an awareness attribute that is related to the possibility that the object has of making the environment aware of its condition. From a communicative point of view this means that each OAW agent must be able to send to other agents information that they may use or not. A reaction by the agents receiving an awareness message is not expected. The description in terms of the interoperability language is as follows:

$$\text{Int\_condition AND ((X?message) AND (Msg\_condition))*--->}
\ll[X!\text{tell(<variable:value>*), may}]$$

with the following meaning: if a condition on the internal state of the OAW is satisfied (Int\_condition) AND the OAW receives one or more messages (X?message) AND each message satisfies the associated condition on its content (Msg\_condition), then the OAW sends (concurrently) one or more messages to

---

1In this sequence messages can be combined using both the AND and OR connectors. In the first case the condition is satisfied only if all of the messages are received, in the latter the reception of one message is enough to trigger the events of the second part. The operator $\ll$ denotes concurrent activation.
other OAWs. Either the Int_condition or the set of incoming messages can be empty, but not both of them; the outgoing messages of the awareness conditions are sent with the may value of the ack parameter.

Let us remark that OAWs do not have an attribute of type 'coordination'. The motivation relies on the fact that OAWs do not have per se, an autonomous behaviour, rather they are just contributing to the definition of the overall C2M. Then it is meaningless to coordinate them with other C2M behavior. Instead, they are able to monitor their modifications and notify them as expressed by the attribute 'awareness'. In other words, they posses a low degree of reactivity.

Each OAW can be created by an enabled Role (the one in the attribute defined by each OAW) belonging to the Organisational Context. This can be done using the primitive overwrite of the interoperability language. The OAW receives the following message:

<Role>?overwrite(empty, Attribute*)

The effect of this message is the definition of the attributes of the receiving OAW which are listed in the second parameter of the primitive.

Furthermore, the definition of an OAW can be modified (adapted by) while the C2M is running: this can be still done using the primitive overwrite:

<Role>?overwrite(<Attribute, NewAttribute>*)

In this way the definition of the Attribute in the first parameter is modified according to the definition contained in the second parameter. Only modifications coming from the role specified in the field adapted by are accepted by the objects.

When the type of the above attributes involves a policy, then before the described communication the related rules have to be applied or the related C2M has to be activated. This second case will be described in section 2.4.4. The same consideration applies, from now on, in all situations in which a policy is specified.

The communication that cannot be described uniformly for all the OAWs involves the attributes considered in the following points.

(1) Assumption of a Role by an Actor

The most important communication for an Actor is related to its assumption of a role. The Role in the attribute assigned by of the Actor, following a policy (which respects the precepts of the role), decides to assign the Actor to a Role. Then the actor is prepared to receive a message of the form:

Role?perform(update, assigned_to:actor)
After this events the Committed-Actor assumes all the responsibilities of the Role. From this event on, the semantics exploits the [parameter] in the message exchanges, which when the message is sent to a Role is specified as follows:

\[ \text{Role}![[\text{actor}]]<\text{message}] \]

[actor] is a parameter that can be empty. When a role receives a message, there are three possible situations:

1) the parameter [actor] is specified, but its content is not in the list of the actors that have assumed that role: in this case the Role sends an error message and the communication fails, unless this communication is a consequence of an explicit choice of a user involved in the handling of this error.

2) the parameter [actor] is specified and its content belongs to the list of the actors that have assumed the role: in this case the role delivers the message to the specified actor.

3) the parameter [actor] is not specified: the role will apply some criteria to decide to which actor, among those playing the role, to deliver the message. This could possibly involve the user, in some crucial cases.

**2) Activation and participation to a C2M**

The Role has to communicate with the C2Ms it is responsible for. To start a C2M it sends a message

\[ \text{C2M}!\text{perform(activate)} \]

A Role can activate all the C2Ms it is responsible for, plus a set of C2Ms that can be classified as publicly available, e.g. the C2Ms describing conversations.

The C2M will take care of asking to the activating Role all the required info: for example in the actual case, the Actors that will play the Roles involved in the C2M.

The C2M, as an agent, will send to each participant involved in itself

\[ \text{Role}!\text{perform(participate, Role') } \]

where Role’ is the Role which started the C2M.

Furthermore, the Role is prepared (by means of its UI agent) to wait for a message from each C2M it is involved in

\[ \text{C2M}\text{?perform(participate, Role')} \]

**3) In-triggers**

The In-trigger attribute appearing in the Task and in the PerfAction represents an incoming communication that carries information that must be received before the Task or the PerfAction can start. This communication is simply described as

\[ X?\text{tell(<variable:value>*)} \]

---

1 Even if in the considered examples only roles can be involved in protocols, the architecture can be easily extended to consider a generic agent if needed.
(4) Activation and accomplishment of a Task by the responsible Role

The Role has to communicate with the Tasks it is responsible to activate and accomplish it. To activate a Task, the Role sends a message of the form:

Task!perform(activate)

(If the Task needs some initial parameters to start the computation, it will ask them to the responsible Role: this will be its first operation)

A role responsible for a Task will be waiting for a message:

Task?tell(return_parameters:value)

where return_parameters is a list containing the result of the computation; if it is empty, by this message the Task simply pass back the control to the Role.

After receiving this message, the Role applies the Criteria of accomplishment.

(5) Activation of an Activity by a Task

The activities realising the Task must be activated:

Activity!perform(activate)

As for the case Role-Task, the Activity will ask the input parameters as first Action. When an activity is over, it must pass back to the Task the results of its computation

Activity?tell(return_value:value)

where return_value is a list

(6) Activation of an Action by an Activity

The relation Activity-Action is similar to the relation Task-Activity: the Action starts with a message

Action!perform(activate)

Then it returns the control to the Activity with a message:

Action?tell(return_value:value)

Before activating an Action, the Activity asks to the initiator Actor if the Action must be delegated to another actor with a message

Actor!ask(doer)

If the doer is different from the initiator, the activity updates the Action attribute

Action!perform(update(doer:actor))

Furthermore the initiator Actor must inform the doer Actor that s/he has to do the Action, thus the initiator actor sends to the doer the message
Actor!perform(update(doer:action))

(7) Assessment and Approval of a Task
When a Task has been accomplished, it must notify this event to the Roles which assess and approve it:
Role!tell(return_value:value).
Then the Task waits for an assessment and an approval:
Role?tell(assessment:value) and (Role?tell(approval:value))

The communication between the Task and the Roles which assess and approve it, possibly happens before the communication between the Task and the Responsible Role to give back the control (4).

(8) Supervision of a Task
When the execution of an Activity or an Action raises problems that cannot be solved by accessing the appropriate system information (lack of specification, errors in Actions, etc.), the Committed-Actor can activate an Interaction with the Responsible of the related Task, either by a simple Interaction with its User Interface agent (yet another bit of its behaviour) or activate the related policy. Typically, the interaction takes the format of an ask(<variable>*).

(9) Management of the Resources
When a Task needs a Resource, it must require its allocation to the Role which is responsible for the Resource. Before doing that, the Task asks the Resource about the responsible role:
Resource!perform(read, governed_by1)
Then, the Task requires the Resource allocation to the Role
Role!perform(update, list:Task))
(where list is a variable containing the list of tasks requiring the resource) and then waits for a message from the role
Role?perform(update, has_allocated: resource)2
After having used the resource, the Task releases it:
Role!perform(update, use: no-more))
where use is a variable owned by the Role, containing the Task which is using the resource (the Role maintains a similar variable for each Resource it is responsible for). It is always the Responsible role which updates the attribute Allocated to of the resource.

Once a Resource is allocated to a Task, the Actions constituting the Activity realizing the Task can use it. This is achieved by a sequence of communication

---
1 governed_by is the attribute of the Task
2 has allocated is the attribute of the Task
that checks for the availability of the Resource (state=NOT in-use) and in the positive case, acquires the Resource for its execution. Finally, the Action releases it when it is terminated. The sequence of communication mimics the one described for Resource allocation.

4.2 The behavior of the Active Artefact

The behaviour of the AA has to be defined with respect to read/update requests, coordination and awareness. The behaviour of the AA can be described as a demon that listens to the environment to catch read/update requests and incoming messages (In-trigger in the coordination attribute) that trigger some internal function and additional communication, and, concurrently, tests conditions on its internal state.

Read and update requests can be described in terms of the interoperability language as the receipt of a message by the AA as follows, respectively:

\[ \text{X?perform(update, <variable:value>*\}) \]

\[ \text{X?perform(read, <variable>*\}) \]

The coordination attribute of the AA describes its capability of receiving from agents and conveying to agents information which is compulsory to coordinate their work. In particular the AA performs this job in two ways: reacting to the fact that a particular condition is fulfilled by sending a message, and activating an internal function as a consequence of the receipt of a message. Each one of these ways can be described in terms of the interoperability language.

As for the first one, the format is as follows:

\[ \text{Int\_condition AND ((X?message) AND (Msg\_condition))}\] \[\rightarrow \text{||[X!tell(<variable:value>*)]}\]

with the following meaning: if a condition on the internal state of the AA (that is on the values of its data-frame) is satisfied (Int\_condition) AND the AA receives one or more messages (X?message) AND each message satisfies the associated condition on its content (Msg\_condition), then the AA sends (concurrently) one or more messages to other agents (X). In this case either the set of incoming messages or the internal condition can be empty.

The second way to perform coordination can be described as

\[ \text{Int\_condition AND ((X?message) AND (Msg\_condition))}\] \[\rightarrow \text{function}\]

The left part can be described as above unless in this case the Int\_condition can be omitted, whereas there must be at least an incoming message. The function triggered is a computation which involves the data-frame of the AA.

The communications related to the coordination is characterized by the fact that the ack parameter has the must value both for incoming and outgoing messages: in fact, coordination implies a compulsory reaction by the agents involved in the communication as well as by the AA.
The awareness attribute of the AA can be described in term of communicationis as done for OAW agents.

4.3. The behaviour of Proctors and C2M

The behaviour of a Proctor is obtained as the outcome of the joint behaviours of the OAWs appearing in its definition. A Proctor can be described as a compound agent, obtained from the composition of agents corresponding to the OAWs.

The composition of OAW within the Proctor agents is defined through a partial order relation capturing the relations among them. The current operational semantics of Ariadne exploits the operational semantics associated to the relational structures (graphs, AND-OR graphs, Petri Nets) in order to construct the operational semantics of a Proctor from the one of its OAWs and relational structure. One could take a "full communication" approach and represent also the relations among OAWs by means of interactions. This approach is taken, e.g., in (Dam, 1994) to deal with adaptability of C2M, and could easily be adopted in the definition of the operational semantics of Ariadne. This however has not been done yet for two reasons: firstly, the relational structure approach is more suitable to describe the behavior of C2Ms to their designers; secondly, the implementation environment (see Chapter 9) is based exactly on the notion of relational structures. The "full communication" approach will be considered in a future version of the semantics of Ariadne when malleability will be supported by formal tools.

As mentioned in section 2.3.2, the behavior of a C2M is the parallel composition of the behaviors of the related AA and Proctor agents on the basis of their communication capabilities.

4.4. Interoperability among C2Ms

In this section we relate the IL primitives to the three linking modes among C2Ms and describe the behaviour of the C2M interfaces.

The first mode is subscription: in this mode a C2M can make the behaviour of another C2M part of its own behaviour. This mode is related to the messages perform(activate) and perform(participate,...). In particular a C2M can make the behaviour of another one part of its own by activating it or having one or more Roles involved in it. These situations are described in the first part of the attribute Subscription of the C2M Interface. They can be realised by the occuring of the following communications in a Role belonging to the current C2M:

\[
\begin{align*}
\text{C2M}!\text{perform(activate)} \\
\text{C2M}?\text{perform(participate,...)}
\end{align*}
\]

The second part of the same attribute describe the symmetric situations: a C2M is called to be part of the behaviour of another C2M and then must involve Roles
belonging to other C2M to participate in itself. These situations are realised in the following way:

\[
\text{C2M.Role?perform(activate)} \\
\text{C2M.Role!perform(participate,...).}
\]

The *inscription* mode, by which a C2M can give or get information about its current state or about the state of the target C2M respectively, is related to the primitives *tell, ask, perform(update,...), perform(read,...) and synch*. The attributes *coordination* and *awareness* of the C2M Interface describe the exchange of information that can occur in this mode: some information can be sent or received requiring or not a reaction, that is an acknowledgment from the receiver; these situations can be realised in the obvious way, using the primitives mentioned above and the correct value of <ack>.

A C2M Interface also takes into account the possibility that some Actions and Tasks belonging to different C2M are executed respecting pre- and post-conditions. This possibility is described in the attribute *Control* of the C2M Interface; it is realised by the receipt, in all the C2Ms containing some of the Actions/Tasks to be synchronised, of the message

\[
\text{C2M.Interface?synch(C2M.Proctor.Action/Task*, <pre-, post-conditions>)}
\]

The *prescription* mode allows a C2M to overwrite the target C2M behaviour in its nominal or actual representation. This possibility is delineated in the *Prescription* attribute of the C2M interface. The modification of the nominal representation of a C2M requires a change in the type of some of the attributes of a component of the C2M: this is realised by the primitive *overwrite*, in particular by the receipt, in the target C2M of the following message

\[
\text{C2M.Role?overwrite(Attribute*, NewAttribute*)}
\]

As stated above, the particular case when the first parameter has the special value ‘empty’ corresponds to the definition of a new set of attributes.

The modification of the actual definition of some component of a C2M is obtained as an enforce of some particular values into some of the attributes defining the object: it can be realised as the receipt of the following message by the component to be modified:

\[
\text{C2M.Role?perform(update, <Attribute: NewValue>*).}
\]

5. A working example

In this section we will consider the field study described in (Carstensen, 1994; Carstensen et al., 1995), formulating a set of requirements for computer based mechanisms supporting the articulation work involved in registering, diagnosing and correcting software errors. These requirements are mainly derived from the findings of a field study of a large software development project at Foss Electric.
The case study has already been described in details in Chapter 1. Here we want to recall that the roles that are relevant for registering, diagnosing, and correcting software bugs are: the tester, involved in the actual testing of the software in the S4000 instrument; the spec-team (ST), responsible for diagnosing the bugs and deciding how to handle the correction of the bugs; the responsible designer (RD), in charge of the correction and the platform master (PM), one of the designers in the project, responsible for verifying the corrections made by the designers.

2.5.1 Definition of a grammar

The first step in the creation of the computational coordination mechanism described in the previous section is the definition of a grammar. Let us consider the definition of a grammar called WF_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 (De Cindio et al., 1982).

WF_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 behavior of the involved roles.

\[
\text{START ::= WF-SA-net}
\]

where \(WF-SA-net = [n_1: SM_1 \ || \ ... \ || n_n: SM_n]\)

\(||\) is the parallel composition based on the synchronization of sending/receiving messages and the labelling functions associated to the names \(n_i\) and to the constituting State Machines \(SM_i\) are as follows:

- \(L_i : \text{Trans-nodes}_i \rightarrow \text{A}_i\) and \(\text{A}_i \in \text{Interactions} \cup \text{T}asks\)
- \(L : \{n_i\} \rightarrow \text{Role}\).

2.5.2 Definition of C2M

Considering the case study, it is possible to develop a C2M based on the previous grammar, that we will hereafter call Bug-Handling (BH). Each component of the C2M is associated to a role. So we will have four components representing the procedures that must be followed by the four involved roles.

The artifact of the BH is the bug form, used by the different roles to convey information at the different stages of work.

The bug form is described in fig. 2.5.1, with additional information on which role, following the conventions stipulated in the group, is supposed to fill in which field.
Figure 2.5.1 The bug report form, with indication of which role is supposed to fill in which fields

The information conveyed by the bug report form constitutes the data frame of the active artefact. In figure 2.5.2 we present the AA

---

**Active-Artefact:**

<table>
<thead>
<tr>
<th>Data Frame: .....</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Access Rights:</strong> {Tester-BH, ST-BH, RD-BH, PM-BH} (read,update)</td>
</tr>
</tbody>
</table>

**Update/ Read Requests**

- X ? perform (update (info))
- X ? perform (read (info))

**COORDINATION**

- Tester ? perform (update (newbug)) --> ST ! tell (newbug)
- ST ? perform (update (STinfo)) --> RD ! tell (STinfo)
- PM ? perform (update (correction:_.)) & (correction=notOK) --> ST ! tell (correction:_.)

**AWARENESS**

- ST ? perform (update (state:_)) & (state:=rejected) -->
  - Tester ! tell (state:_ `may`
- ST ? perform (update (STinfo)) -->Tester ! tell (state:_ `may`
- clock? tell (signal) AND not(RD ? perform (update (RDinfo))) -->
  - RD ! tell (solicit) `may`

---

Figure 2.5.2 The Active Artefact of the Bug Handling
In the first part of the BR, the access rights to the artefact are specified. This means that all the proctors of the BH can send to BR a request to perform both a read and an update.

Moreover, coordination and awareness are specified. Awareness and Coordination has the form:

\[
\text{condition} \rightarrow \parallel \text{event}^* \\
\text{as explained in details in section 2.4.2.}
\]

For example, in the BH we want to specify that when the BR receives from the ST info on the bug to be corrected, it informs the Tester, in awareness mode, if the bug that s/he reported has been rejected.

The corresponding awareness is:

\[
\text{ST?perform (update(state: \_)) } \& \text{(state= rejected)} \rightarrow \text{Tester!tell(state: \_)}
\]

As explained in section 2.3.2, each C2M is constituted by agents, called Proctor, capturing the behaviour of the associated procedures. In our example, following the grammar, each proctor can be a composition of Tasks and Interactions.

Designing proctors, it is possible to associate conditions to arcs.

If a user wants to specify the conditions under which a message must be delivered, s/he specifies the conditions as a label of the arc entering the transition corresponding to the sending of the message.

For instance, let us suppose that we want to specify that the message of the considered example must be sent by the responsible designer to the BR only if \( \text{var1} \) has value \( \text{val1} \). The corresponding net portion will be:

![Diagram](image)

Figure 2.5.3 Conditions on arcs

If no conditions are related to an arc, the corresponding transition is always authorized.
From another perspective, conditions can be used in order to specify alternative behaviours arising in relation to a different content of the same message. For example, let us suppose that the RD acts differently if s/he receive a bug to correct that is classified as catastrophic. For example, if the error is not catastrophic the regular task "correct" is to be activated. Otherwise, the RD must activate a task "URGENT correct", as illustrated in figure 2.5.4:

![Diagram](image.png)

**Figure 2.5.4 An alternative behavior**

To summarize, in Figure 2.5.5 we describe the overall design of the Bug-Handling
Following is the explanation of the proctors and the active artefact of BH (numbers refers to transitions in figure 2.5.5).

-1- After having detected an error, the tester requires the BR to perform an update of its internal values. Here ‘new bug’ stands for a list of the type <slot name: slot value>*; indicating which characteristics of the BR the tester is initializing (according with the information in figure 2.5.1). On this event the BR reacts with a message to the spec-team (-4-), communicating that a new bug has to be processed, conveying all the information provided by the tester.
-2- The tester requires to HelpRequestConv (a predefined conversation protocol) to activate itself. As explained in section 2.4, the activated C2M can ask for needed information. In square brackets the mandatory response is indicated, meaning that the ST has to be involved as partner. This is an example of subscription to another C2M.

-3- The BR informs the tester on the evolution of the bugs it reported. This is only a sort of notification that does not impact on the behaviour of the tester.

-5/6- The spec-team requires the BR to update its values according to its new decisions. In the first case, the spec-team requires only the change of the bug state, while in the second one, i.e. bug accepted, more information is conveyed. As before, ‘STinfo’ stands for the list provided by the spec-team. On this event the BR reacts with a message to the responsible designer (-10-), communicating that a new bug has to be processed, conveying all the information provided by the spec-team.

-7- If the correction is not approved by the platform master, the BR sends to the spec-team a message indicating the necessity to re-process the bug.

-8- The Spec-team can, at every moment, be involved in a conversation with a tester asking for help.

-9- The RD can receive from the BR (last communication in the awareness section) a solicit when the time allocated for its work is almost expired. Solicit is supposed to be a variable that is associate with every proctor.

-11- The RD requires the BR to perform an update of its internal values.

-12- When the PM starts its work, it asks BR for the needed information on the bug.

-13 After having verified the bug, the responsible designer requires BR to perform an update of the value of the slot 'correction'.

The transitions that are not numbered correspond to actions/tasks in the FoW.

Let us now consider the relations of the BH with its external world represented by the C2Ms interoperating with BH, namely a Scheduler (of the software activities), a SW-C2M (managing the software decomposition into modules), and finally the SW Process.Manager (SW-P.M), that is, the component of the C2M SW Process describing the behavior of the Manager responsible for the task devoted to changing the bug report process. These relations are described in the BH-Interface as following:
BH-Interface=

<table>
<thead>
<tr>
<th>name</th>
<th>BH</th>
</tr>
</thead>
<tbody>
<tr>
<td>coordination</td>
<td>BR:ST ? perform (update (STinfo)) -&gt; Scheduler! tell(new-bug)</td>
</tr>
<tr>
<td>awareness</td>
<td>BR:ST ? perform (update (STinfo)) -&gt; SW-C2M! tell (STinfo) 'may'</td>
</tr>
<tr>
<td></td>
<td>BR:PIM? perform (update (correction: OK)) -&gt; SW-C2M! tell ((correction: OK)) 'may'</td>
</tr>
<tr>
<td>subscription</td>
<td>tester-BH:HelpRequestConv! perform (activate), [ST-BH]</td>
</tr>
<tr>
<td></td>
<td>ST-BH: HelpRequestConv? perform (participate, tester-BH) *</td>
</tr>
<tr>
<td>access rights</td>
<td>none</td>
</tr>
<tr>
<td>control</td>
<td>Scheduler ?</td>
</tr>
<tr>
<td></td>
<td>synch((Scheduler...AssignPIM, BH.PIM-BH, (BR ! read(bug-to-correct))), sequence)</td>
</tr>
<tr>
<td></td>
<td>Scheduler ! synch ((Scheduler..AllocateResources, BH.RD-BH, (correct))), sequence)</td>
</tr>
<tr>
<td>prescription</td>
<td>SW-P.M? over-write (empty, &quot;see figure 4&quot;)</td>
</tr>
<tr>
<td></td>
<td>SW-P.M ? over-write (BR, tester, PIM, new-BR, new-tester, new-PIM)</td>
</tr>
</tbody>
</table>

The meaning of the various slots are as follows:

coordination: When the Spec-team accepts a bug, the BH must tell the Scheduler that a new correction must be considered during the planning.

awareness: Moreover when the new bug is accepted, the BH notifies the SW-CM, so that the latter is aware that a bug has been detected within a specific module.

The same happens when the bug is corrected

subscription: to express that BH makes use of the Conv-CM.

The above slots are automatically derived from the definition of the protocol given by the designer. The following ones have to be explicitly given by the designer when defining the interface:

access rights: No CM has access rights on the AA of the BH.

control: In the first communication the BH is passive, in the sense that the PM is suspended until the Scheduler performs ‘AssignPM’ and informs the BH. In the second one the BH must require the activation of ‘Allocate Resources’ before ‘correct’ can be performed by the RD.

1 sequence is a shorthand indicating a relation imposing a sequential execution of the involved actions that arises from a set of pre and post conditions. This means that the postconditions of the action that must be performed as first are a subset of the preconditions of the second one, and so on.
prescription the BH is prepared to receive from the SW-P.M two types of messages: one for its definition and one for its adaptation (In the interface we give only an example of possible over-ruling).

2.5.3 The bug report and the binder

Until now, we have considered only the handling of the single bug (let us recall figure 2.5.5 where we show the protocol stipulating the behaviour of the involved roles ). The protocol could be better specified in order to adhere completely to the description of the field study in (Carstensen and Sørensen, 1994).

As pointed out in (Carstensen and Sørensen, 1994), there are three problems that must be faced when considering the bug correction: ensure that all registered bugs are treated until they reach a final status, handle a process of distribution, compilation and, finally, ensure that a up to date state of affairs overview is provided to everyone involved in the bug handling process.

Is it self-evident that the protocol in Figure 2.5.5 does not support in any way this last point. In Foss Electric the overview of the state of affairs was provided by the binder. All registered bug are filed in a binder accessible to all testers and designers. One of the software designer 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.

We propose an architecture with two distinct C2M: one for the handling of the single bug (BH), and one for the binder (binder-C2M).

The binder-C2M is a rather simple one, constituted mainly by an AA plus an elemental protocol that provides the "interface" with the external world. Thinking about a computational mechanism, many of the "tasks" that are described in the field study are unnecessary. For example, in real life it is necessary to provide a copy of the bug report form, and a distinction between the real one and the copy must be supported. With a computational artifact this is no longer required, or at least it is not necessary that the central file manager (hereafter CFM) takes care of it.

In the following we will roughly describe the binder-C2M together with the changes that must be performed on the BH.

The AA of binder-C2M represents the actual binder of the case-study. In its data frame there is a slot called "repository", where the names of the actual bug forms are memorized and organized. Here we have only names that are organized following the required criteria. The acquisition of the information is realized through communication. This means that there is a inscription between the Binder-C2M and the BH that can be expressed in the following way:

BRi ! ask(bug report)

where "bug report" is a list of pairs (slot name: slot value), and slot name belongs to the data frame of the BR (i.e. the active artefact of the BH). This
communication allows the Binder to access the information on a specific bug whenever it is needed.

The corresponding communication must be inserted in the BH.

How the repository is organized depends on its intrinsic complexity: sometimes a tree would be needed, while for example in this case a record seems enough.

So, in the binder we have a data structure of the form (see figure 2.5.6):

\[(\text{entry} : (\text{BRi})^*)^*\]

where each entry corresponds to one of the entry of the actual binder (Non-corrected catastrophic, non-corrected essential and so on). Moreover we added an entry where all the bug reported by testers are collected (it is possible to consider with no additional problem any different classification scheme). As pointed out in (Carstensen and Sørensen, 1994), "...these entries reflect 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...."

**Active-Artefact:**

<table>
<thead>
<tr>
<th>name</th>
<th>Binder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Frame:</td>
<td>.....</td>
</tr>
<tr>
<td>Repository:</td>
<td></td>
</tr>
<tr>
<td>New</td>
<td>(BR1, ...BRn)</td>
</tr>
<tr>
<td>Non-Corr Catast.</td>
<td>...</td>
</tr>
<tr>
<td>Non-Corr essential</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 2.5.6: The Active Artefact of the binder-C2M

Since the main aim of the binder is to provide an overview of the state of affairs, the AA must be prepared to reply to request coming from a set of specified roles. This means that the following behavior is to be associated to the binder (Figure 2.5.7):
Let us now describe the behaviour of the CFM (Figure 2.5.8).

The first elemental protocol describes, roughly speaking, the job of a repository manager: to receive information on the elements to be put in the repository (in this case the bug forms), to organize them following some internal rules and, finally, to update the repository.

The second protocol describes the CFM in its interaction with the PM (the one in the binder-C2M). Informally, when it is time to start the correction verifications, the CFM reads in the Binder which corrections must be verified and then tell this info to the PM. The arrow coming from nowhere graphically indicates a synchronization with another component. In the binder-C2M interface this will be better specified (The CFM must receive by the Scheduler the name of the PM for the specific platform.).

The PM is defined as follows (Figure 2.5.9):
The rule that allows to the PM to decide which transition to activate is "If there are no more correction to verify then DONOTHING, else tell the PM-BH one of the remaining corrections."

Other changes are necessary in the C2M describing the handling of the single bug form. As a first thing, it is necessary to take into account that now every role has the possibility to ask for information on the overall situation. For this reason, we add to every elemental protocol a parallel behaviour that describes this possibility (Figure 2.5.10).

Moreover, the coordination of the AA must be modified, in order to add the coordination with the binder (Figure 2.5.11).
COORDINATION

Tester ? perform (update (newbug)) --> ST ! tell (newbug) || Binder! tell(info)

ST ? perform (update (STinfo)) --> RD ! tell (STinfo) || Binder! tell(info)

RD ? perform (update (RDinfo)) --> Binder! tell(info)

PM ? perform (update (correction: _)) & correction= notOK-->
ST ! tell (correction: _) || PM-Binder ! tell (correction: _)

PM ? perform (update (correction: _)) & correction=OK-->
PM-Binder ! tell (correction: _)

Figure 2.5.11: The new coordination of the BR

The PM proctor must be modified, so that it waits for a message about a new correction to verify coming from the PM_BinderC2M, i.e. the part of the PM behaviour related to the binder (Figure 2.5.12). Of course, the message for the synchronization with the scheduler must be cancelled by the Interface of the BH.

PM-BH

Figure 2.5.12: The PM-BH

In a certain way the behaviour of the PM as a whole could be described as in figure 2.5.13. A platform master (described at the binder level) gets a list of \( n \) corrections to be verified and it activates \( n \) instances of the platform as described in the BH (i.e. the behaviour of the platform master as it handles the single bug).
6. The primitives of the notation in the agent perspective

ARIADNE provides a set of primitives to manage the definition, modification and use of the C2Ms.

Some of these primitives can be put in relation with the communication described in terms of the Interoperability Language.
The primitives provided of the $\gamma$-level are the Mall and the Link functions.

The Mall function allows the creation and modification of the grammar used to specify the protocols. Mall takes a grammar as an argument. If the argument is empty then Mall denotes a creation, otherwise it denotes a modification.

Mall can act on the definition of some of the basic elements of the grammar (that is on the definition of the OAWs, the AA and the Relational Structures) or on its semantics.

The modification (or the definition) of the OAWs or of the AA can be realised as the receipt, by the corresponding agents, of an over-write message; the agent to be modified (or defined) receives from the User Interface of the appropriate role the following message:

UI-Role?over-write(Attribute*, NewAttribute*)

Clearly, if the first parameter is ‘empty’, then this message corresponds to the definition of the object.

By the Link function it is possible to build a composed C2M from two or more existing ones. The form of the Link function is the following

Link(M1,...,Mn, I1,...,In)

where $M1,...,Mn$ are C2Ms and $I1,...,In$, their interfaces. The construction of a compound C2M by a Role (?) corresponds to the definition of the Interfaces of the component C2Ms: thus the Link function is realised as the receipt by all the involved Interface agents of an over-write

UI-Role?over-write(Attribute*, NewAttribute*)

The primitives of the $\beta$ level are: access, define_C2M, modify_C2M, animate, simulate, build_history and access_history

The primitive access simply allows one to use the grammar defined at the $\gamma$ level to create C2Ms: it does not require any communication among agents.

By the primitive define_C2M(X) it is possible to define a new C2M, named X, using the available grammars; an existing C2M X can be modified by the primitive modify_C2M(X., modification_type). The parameter modification_type contains information about which component of the C2M has to be modified and what kind of modification must be performed. These two primitives can be realised as a communication sent from the user interface to current C2M, which receives the following message

UI-Role?overwrite(Attribute*, NewAttribute*)

The attributes refer to the component of the C2M appearing in the parameter modification_type; also in this case if the first parameter is ‘empty’ the message realises a define_C2M, otherwise a modify_C2M.

The primitives animate(X) and simulate(X), which provide the user with the possibility of ‘playing’ with the C2M, are realised by the following message
UI-Role?perform(activate) received by the C2M. The parameter needed to start the simulation will be required by the C2M to the User Interface.

The primitives build_history and access_history are managed by the notation in relation with the environment it is embedded in; they do not involve the agents.

Finally, the $\alpha$ level provides the primitives: access, define_instance, modify_instance, make_permanent, activate, enforce, build history and access_history.

The primitive access simply allows one to use the C2M defined at the $\beta$ level to create its instances: it does not require any communication among agents.

The primitive define_instance($X,Y$), where $X$ is the C2M name and $Y$ the instance name, is used to create an instance of an existing C2M: it is realised as a set of messages

UI-Role?perform(update, <attribute, value>*

received by all the components of the C2M.

The primitive modify_instance($Y$, modification_type) allows for structural modification of the instance: it is realised as an over-write:

UI-Role?overwrite(Attribute*, NewAttribute*)

received by the suitable agent in the current instance. This modification can be saved in a new protocol, by the primitive make_permanent($Y$, NewName): this primitive create a new protocol named NewName, which $Y$ is an instance of. It can be realised by means of a primitive, like in the case of define_C2M, from the current instance instead of the User Interface.

The primitive activate starts the execution of the current instance: it corresponds to a

UI-Role?perform(activate)

received by the current C2M instance.

The primitive enforce($X$, NewConfiguration) can be used when the instance has been activated. Its invocation allows for an instance behaviour to proceed from new configuration. This effect is obtained sending from the UI a set of messages

$X$!perform(update,<attribute*, value*>)

to each component $X$ whose state has to be modified.

The primitives build_history and access_history, as for the $\beta$ level, are managed by the notation in relation with the environment it is embedded in; they do not involve the agents.

. Achievements and future Work

For the purpose of designing computational coordination facilities that provide support for cooperating actors in managing the complexity of articulating their
distributed and yet interdependent activities, a multi-agent architecture has been proposed as an effective means to the design of computational mechanisms of interaction. From the evidence of the body of empirical studies of uses of artifactually embodied protocols for articulating cooperative activities, we have derived the set of constitutive agents and a characterization of their communication needs. Indeed, the former corresponds to the objects of articulation work and to how they are combined to construct C2Ms at any level of complexity, while the latter contribute to the realization of the functional primitives the mechanisms should provide in order to become a dynamic support for coordination. Specifically, the attention has been focused on the primitives devoted to dynamic reconfiguration, cooperative control of propagation of changes, and interoperation among multiple C2Ms. Accordingly, the agents’ Interoperability Language is constituted by primitives representing various modes of linking C2Ms: inscription, subscription and prescription.

At the present stage, the proposed architecture is not defined in all the needed details. For example, more investigation is needed in order to express how the ‘activate’ and ‘over-write’ commands can actually be executed on the existing agents. This investigation could suggest ways to to enrich the architecture with specialized agents devoted to these ‘meta-activities’ and with the concomitant part of the Interoperability Language. However, the approach seems to be very promising.

On the one hand, the current multi-agent architecture allows us to cope with requirements that were not covered by the previous formulation of the notation, and which have been be integrated in this latter since it deals with the orthogonal aspects of designing the internal structure and behaviour of the agents.

On the other hand, the current multi-agent architecture constitutes a sound basis on top of which the missing aspects can be added, in a layered and modular way. These aspects relate not only to the refinement of the primitives of the current Interoperability Language but also to the addition of more complex functionalities exploiting the underlying model and the communication flow to cope with the problem of designing C2Ms that maintain their coordination capabilities in face of partial and possibly inconsistent information, and to manage the time dimension, that is still not covered by Ariadne.

**Acknowledgments**

The authors want to thank their partners in the COMIC project. A special thank is due to Kjeld Schmidt for the many discussions that led to the refinement of the notation and of the multi-agent architecture. We are indebted to Peter Carstersen for his help with the working example.
References


Chapter 3

SOS as an implementation framework for the C-MOI notation

Leandro Navarro
Universitat Politècnica de Catalunya

The SOS is an architecture developed in the COMIC project to provide a rich computational environment for large and complex CSCW applications. This chapter describes how that architecture relates to the C-MOI notation. One realisation of that architecture is Aleph. The most important features of Aleph are the Resource Manager, the Aleph-Tcl notation and the mechanisms for linking environments (contexts and federation managers). Aleph-Tcl is a notation, an interpreted language to manipulate, configure, adapt and write new and existing Aleph components. It is compared with the C-MOI notation. Finally, the I*EARN network organisation is described in terms of the C-MOI notation, as an example of a real world environment where Aleph is being introduced.

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 Strand4 Sharing Objects group 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 common set of primitives, a common set of services including support for articulation work (Simone and Schmidt, 1994) (Malone 92).

The S*S provides functions at the semantic level of cooperative work (both work and articulation work), some of which may be 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 build on existing ones.

The S*S provides mechanisms, in the form of Manager objects, that stipulate and mediate both, work and articulation work, therefore reducing the complexity of developing new tools to support cooperative work.
The S*S architecture is modular and interoperable because 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 available at any organisation running an S*S based environment, 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.

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 artefacts, 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.

For example, in the Aleph 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 SOS 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 may be navigate, reserve, move, name, consume, characterise, relate, activate, join, leave, enroll, define category, classify, etc. (under development).
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 modularity, flexibility 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 artefacts 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.

2. The Aleph System

The Aleph system is a partial implementation of the S*S architectural model. It is intended to be a test bed for the testing and demonstration of the architecture with special emphasis on the issues raised on the COMIC Project regarding large scale social interaction environments. The main ideas are to support:

- **malleability.** This feature enables users in making global and permanent changes to the behaviour of components. Actors have facilities to change, specify and re specify the behaviour of the mechanisms either directly, by invoking the configuration primitives through control panels, or indirectly, by doing actions and therefore sending events producing changes in their environment.

- **scalability.** This means supporting a large number of inter-related tasks sharing resources, supporting diverse user interaction mechanisms, and supporting cooperative interactions across diverse and distributed systems, locations and organisations.
- **linkability.** Applications such as workflows, calendar, classification tools are all interdependent, and they are all based on a common set primitives and a common implicit notation provided by the Aleph components. Therefore they can be built and linked very easily to form new mechanisms by means of the Aleph-Tcl notation.

- **other system issues,** related to distribution support on the large scale, fault tolerance, asynchronous systems, integration with Internet services, etc.

One of the most relevant components in Aleph is the Resource Manager. It is responsible for the management of resources, trading for interfaces, binding and adapting requests with offers, and handling contexts. *Resource Management* is a function required to orderly access to resources for cooperative work, and any other entity in a organised way (Gimenez et al., 1994). 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 an 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 actors 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.

*Context management* is the function to group entities on domains where work may take place, where events are distributed, where a given policy applies. Domains determine discontinuities where events may be stopped or transformed, where policies may be circumscribed, and they provide a limited view of an otherwise large and complex environment.

Therefore, the Resource Manager is a component specialised in the articulation of the distributed activities of multiple actors and resources in a given field of work.

![Diagram of Resource Manager](image)

Figure 2. The Resource Manager plays a central role in the transition from nominal to actual. The RM knows and applies the responsibilities and policies of the Organisational Context, and keeps information about resources available and in use, actors and roles.

### 3. The Aleph-Tcl notation

The *Aleph-Tcl* notation is a powerful open language that provides primitives to access the functionality provided by several Aleph components. This notation can be easily extended to incorporate the functionality of new additional components. This supports the construction of new components, mechanisms and applications.
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).

The malleability feature of mechanisms of interaction has been defined as supporting users in making some global and permanent changes to its behaviour. We have chosen this extensible interpreted language to support users in making such changes to the behaviour of computational components. This is a desirable feature of the notation, but also a hard software problem. It is possible to send modifications while a script is running (commands sent to a script are executed by the interpreter) and events may also be handled by the interpreter (an event may be bound to a Tcl procedure).

A compromise to simplify the design of the Aleph software has lead to define a level of malleability at the design phase by the choice of which functions are visible, and therefore malleable, at the Aleph-Tcl notation level: parts written in Aleph-Tcl may be respecified by actors while the rest (specified in a regular programming language such as C) may not be changed by any actor (only in future releases of the software by the software development group).

Actors have facilities to change, specify and respecify the behaviour of the mechanisms either (1) directly at the Aleph-Tcl shell, or indirectly (2) by performing tasks, that in form of events change the behaviour of components as consequence of changes in their environment, or (3) by invoking the configuration
primitives through the control panel of each component. Event though the user interface of each component is separate from the Aleph notation, control panels exhibit the malleability and modularity of the underlying notation.

These changes can be done cooperatively, as part of the cooperative effort, while the mechanism is running. Currently there is a primitive support to restrict modifications to a particular role or an actor and to handle concurrent modifications by several actors.

The Tcl language supports radical changes while an application is running. Attributes of an object can be made:

```
Leandro config -assigned_to comic.deliverable.editor
```

or major changes such as redefining a procedure in the Resource Manager object:

```
(send ResourceManager {proc policy.authorize {role resource} { if ...}}).
```

### 3.2. Linkability

The mechanism is based on the ability of Tcl (and Tcl-dp, a distributed version of Tcl) to exchange messages between components. Implicit in the language there are powerful mechanism to relate CMOIs.

The Resource Manager plays a central role to provide primitives to support work articulation with respect to most of the objects of articulation work (OAW). It plays a central role in the C-MOI linking modes that have been identified in empirical studies:

- **reference mode**, the Resource Manager contributes to the definition of any other C-MOI since it is referred when an actor has to assume a role, use a resource or change to another context. The RM complies with the Organisational Context and applies the required policies. Since this mechanism may be potentially different in different environments, this reference makes the referring C-MOI more independent of the particular Organisational environment.

- **awareness mode**, the Resource Manager (or more precisely the Finder) 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 interest to the Event Service to receive Events related to the usage of a given resource. Similar mechanism may be used with roles or actors.

- **accommodation mode**, two components executing workflows generated with two different β level notations (e.g. graphs and Petri Nets) can be linked by means of the public basic concepts of state and action. That information will be
exchanged between components using the Aleph-Tcl notation, and if present, both will use the Sharing Service.

- **over-rule mode**, by means of the *over-write* primitive, any component may be changed, even cooperatively by other components.
- **control mode**, by means of the *synch* primitive, actions on different components may observe a common partial order.

In general, the Resource Manager is responsible for enforcing and applying the organisational policies and restrictions to the objects of articulation work (OAW). In addition, the RM knows and applies the mechanisms, procedures and policies to access to most OAW relieving other mechanisms of the detailed knowledge of the local Organisational Context and any local mechanisms to access OAW. Any C-MOI may reference the RM for those purposes, what increases the applicability of that C-MOI or component to a wider environment, even to federated environments.

In addition, any component can reference the S*S Event Service to advertise, supervise, monitor, be informed of changes on any 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.

### 3.3. Interoperability Language

The aleph Tcl language is basically a Tcl interface to the Resource manager, the Finder, Binder and Context manager. Its basic functionality is to provide the necessary commands to work with these tools. There are four kind of objects: *resources, roles, actors* and *contexts* (Rodríguez 1994; Benford et al., 1994), therefore Aleph commands are structured in four groups depending on what kind of object they relate to.

Based on the above interoperability modes, and the basic primitives of the Interoperability language identified in chapter 1, the following table compares the list of basic primitives of the interoperability language are checked against the Aleph-Tcl notation to look at the implementability of the primitives on Aleph.
SOS as an implementation framework for the C-MOI notation

<table>
<thead>
<tr>
<th>Basic Primitives of the Interoperability language</th>
<th>Realisation in Aleph-Tcl</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tell (&lt;variable : value &gt;*)</code></td>
<td><code>send &lt;Receiver&gt; [variable*]</code></td>
</tr>
<tr>
<td><code>ask (&lt;variable &gt;*)</code></td>
<td><code>set *variable [ send &lt;Receiver&gt; { print $... } ]</code></td>
</tr>
<tr>
<td><code>perform (update (&lt;variable : value &gt;*))</code></td>
<td><code>send &lt;Receiver&gt; { set &lt;variable value&gt;* }</code></td>
</tr>
<tr>
<td><code>perform (read (&lt;variable &gt;*))</code></td>
<td><code>send &lt;Receiver&gt; { print $&lt;variable&gt;* }</code></td>
</tr>
<tr>
<td><code>perform (activate (&lt;protocol&gt;, &lt;participant&gt;*)</code>)</td>
<td><code>send &lt;Receiver&gt; { &lt;protocol&gt; ( &lt;participant&gt;* ) }</code></td>
</tr>
<tr>
<td><code>perform (participate (&lt;protocol&gt;, &lt;participant&gt;*)</code>)</td>
<td><code>send &lt;Receiver&gt; { &lt;protocol&gt; ( &lt;participant&gt;*,&lt;receiver&gt; ) }</code></td>
</tr>
<tr>
<td><code>synch (&lt;C-MOI.Component.Action/Task&gt;*</code>, <code>&lt;partial-order&gt;*)</code></td>
<td><code>send ResourceManager { synch (&quot;&lt;C-MOI.Component.Action/Task&gt;*&quot;, </code>&lt;partial-order&gt;*&quot;) } (not supported)`</td>
</tr>
<tr>
<td><code>over-write (&lt;items&gt;*, &lt;new-items&gt;*)</code></td>
<td><code>send &lt;Receiver&gt; { set &lt;items&gt;* &lt;new-items&gt;* }</code></td>
</tr>
</tbody>
</table>

3.4. Large scale and contexts

The Tcl objects seen below, especially, actors and resources, are clustered in contexts. A context is like an environment, inside of it there are resources, actors (that uses these resources) and a set of roles that restrains who and how each actor can work with the resources in that context. This concept is very useful to deal with problems of scalability and heterogeneity: linking contexts situated in different locations, organisations, systems.

By scalability we mean that the system must be able to grow without affecting the current state of the existing systems. Heterogeneity refers to the problems that occur when two or more systems have to cooperate in order to carry out a certain task. Both problems are closely related to the diversity of large social and computational settings (Schmidt and Bannon, 1993).

In order to enable cooperation among a large and heterogeneous set of entities, we have chosen a federated scheme, where entities decide to cooperate keeping freedom of association and there is no central authority. Little agreement and mutual knowledge between domains is required before interaction can occur partially because every organisation is responsible for itself, it will apply their own management policy and restrictions for the environment they own.

After an agreement between two organisations is achieved, and before the federation is operative, boundaries have to be identified and circumvented with boundary objects and interceptors. In Aleph we address this problem with the Federation managers and the Aleph-Tcl news interface.

4. Aleph-Tcl commands

All the commands about the different elements have been implemented according to an object oriented structuring. So, each element has a Tcl object and a set of methods associated.
Aleph commands are clustered in five groups according to the object oriented philosophy. These groups are: actor functions, resource functions, role functions, context functions and automaton/lookup functions. A detailed description of the commands is available in (Benford et al., 1994).

Here is an example of part of an action illustrating how the procedure to reserve a meeting room may be specified in Aleph-Tcl:

```tcl
print "Looking up for the candidates"
print [lookup "room" "meeting"]
print "Selecting the best (for the organisation) available"
set candidate [automaton "room" "meeting" "gerard" "UPC/comic/str3"]
print "The best one is:" $candidate
print "Connecting"
set name [lindex $candidate 0]
$name connect "gerard"
```

Sometimes it may be necessary that one single entity plays two distinct characters. For instance, we could need an actor to be represented as a resource in order to send mail or to establish a multimedia communication with him. In Aleph-Tcl an object may be redefined. That may be easily done in the following way:

```tcl
... 
actor gerard -name "Gerard Rodriguez"
print [gerard get -roles]
resource gerard -new 10 -name "Gerard Rodriguez"
print [gerard get -policy]
print [gerard get -roles]
...
```

The first command "actor" defines the tcl object "gerard" as an actor named "Gerard Rodriguez", while in the second part, the resource "gerard" is also
created. Therefore, methods for actors, such as "get -roles" or resources, such as "get policy" may be applied to "gerard": that object has several views.

Aleph also provides a function to know the definitions done during an aleph session. This is useful to keep a history of changes, and to distinguish between temporal and permanent changes.

5. Aleph-Tcl as a C-MOI notation

In previous sections, the malleability and linkability of the Aleph system and notation have been examined. Now, the Objects of Articulation Work (OAW) are examined.

All OAWs have in common several attributes: to express the possibility of definition and modification by a role specified in the Organisational Context (Defined by, adapted by), to express the capability of one object to make other objects aware of its progress (awareness). The first two are handled by the Resource Manager, while the awareness attribute is handled by the Event Manager. Currently the Aleph-Tcl notation does not reflect them.

These attributes are not reflected in the following table comparing groups of Aleph-Tcl primitives. This table describes how each primitive of Aleph-Tcl affects attributes of OAW. The notation used is: OAW.attribute (e.g. actor.assigned_to refers to the "assigned to" attribute of actors).

<table>
<thead>
<tr>
<th>ACTOR:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>actor</td>
<td>an_actor [-new] -name description</td>
</tr>
<tr>
<td>an_actor</td>
<td>get-name</td>
</tr>
<tr>
<td>an_actor</td>
<td>get -roles</td>
</tr>
<tr>
<td>an_actor</td>
<td>put -role rolename contextname</td>
</tr>
<tr>
<td>an_actor</td>
<td>delete [-role rolename contextname]</td>
</tr>
<tr>
<td>an_actor</td>
<td>put -incontext contextename</td>
</tr>
<tr>
<td>an_actor</td>
<td>move [-from contexte1 ] -to contexte2</td>
</tr>
<tr>
<td>an_actor</td>
<td>delete [-from contextname]</td>
</tr>
<tr>
<td>an_actor</td>
<td>get -context</td>
</tr>
<tr>
<td>actor -list</td>
<td>actor.all</td>
</tr>
<tr>
<td>(RM)</td>
<td>actor.committed_to, actor.doer,</td>
</tr>
<tr>
<td></td>
<td>actor.assigned_by</td>
</tr>
</tbody>
</table>

actor.personal data, actor.locality, actor.skill
actor.assigned_to
actor.committed_to, actor.doer, actor.assigned by
Several attributes are not visible through the Aleph notation (RM), but the Resource Manager takes them into account. Most of the attributes related to the Organisational Context (responsibilities, commitments, policies) are handled by the RM alleviating the complexity of other components.

For instance, when an enabled role invokes the "resource" command, the resource manager checks the "defined by" attribute for the type of resource being created and updates the attributes of the new object accordingly.

When an actor wants to assume a role, he invokes the "an_actor put -role" command. The RM applies any required policy and assumes that change, updating the role.assigned_to attribute (i.e.: Role?perform(update, assigned_to: actor).

In addition, in Aleph-Tcl is possible to activate any other object (written in Aleph-Tcl or other languages) by writing its path name and arguments. This is used to launch any application, task, C2M, etc. It is equivalent to the...
$X!perform(activate)$ command. Optionally a value may be returned at the end, equivalent to the $X?tell(return_param:value)$ command.

<table>
<thead>
<tr>
<th>SEARCH and SELECT CANDIDATES:</th>
</tr>
</thead>
<tbody>
<tr>
<td>lookup type subtype context</td>
</tr>
<tr>
<td>automaton [-connect] type subtype username context [wished]</td>
</tr>
<tr>
<td>obj_name(resource) connect username</td>
</tr>
</tbody>
</table>

Figure 5.- Aleph-Tcl primitives related to selecting and using resources.

When a Resource is needed, the automaton command (or "$lookup", "$connect" that perform parts of "$automaton") is activated. The RM then looks at responsibilities, applies any required policy, assigns a resource to the caller, updates all required attributes, annotates that event in the accounting records, generates events accordingly, and when the resource is released does the necessary actions.

This is very useful since any mechanism can use resources ignoring how resources must be properly allocated and released, which is a complex and possibly organisational dependent task.

<table>
<thead>
<tr>
<th>CONTEXT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>context a_context [-new] -name nom</td>
</tr>
<tr>
<td>context -list</td>
</tr>
<tr>
<td>a_context get -name</td>
</tr>
<tr>
<td>a_context get -indexes</td>
</tr>
<tr>
<td>a_context get -hierarchy</td>
</tr>
<tr>
<td>a_context get -below</td>
</tr>
<tr>
<td>a_context get -father</td>
</tr>
<tr>
<td>a_context put -index indexname</td>
</tr>
<tr>
<td>a_context delete [-index indexname]</td>
</tr>
</tbody>
</table>

Figure 6. Aleph-Tcl primitives related to contexts.

This provides indexing facilities for grouping and accessing resources in a large environment of one or several (federated) organisations. Any operation is restricted by default to its context. Each RM is responsible for a set of contexts, and one context may be the union of two or more contexts located at different environments, locations, departments, etc. united by the federation manager. This way, the same primitives are used to move from one task to another, than to move from one organisation to another: this is called federation transparency.

Contexts are useful to establish boundaries to the propagation of events, to introduce policies to enter or leave a given context, to decide responsibilities
based on contexts. We have found that contexts are key to the organisational context issues as well as to scalability issues.

6. Conclusions

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 $\gamma$ 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 $\gamma$ 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 $\beta$ notation for that mechanism. A projection of that $\beta$ level notation is the graphic notation provided by the control panel of a mechanism that is used to respecify their $\alpha$ notation.

In addition, particular features of the Aleph-Tcl notation are: transparency of some complex behaviour of OAW (done by the RM) that increases component transportability across organisational boundaries, the use of contexts to group objects, and the mechanism to federate separate environments to work in a large scale inter-organisational environment.

7. An example environment: The I*EARN network organisation

I*EARN is a non-profit global network organisation of about 500 primary and secondary schools in over 20 countries. It is definitely a large scale learning organisation. In their words, their goal is:

*I*EARN empowers teachers and young people (ages 6-19) to work together in different parts of the world at very low cost through a global telecommunications network. The purpose of I*EARN is to enable participants to undertake projects designed to make a meaningful difference in the health and welfare of the planet and its people. I*EARN is a non-profit organisation. I*EARN is expanding to additional international sites daily and now includes over 500 schools in more than 20 countries*.

From their definition, I*EARN is a non-profit organisation that supports worldwide educational projects involving actors from different member organisations, more than an international network of educational resources. It is interesting to note that I*EARN activities involve volunteers who participate in extra-curricular projects: they sometimes do activities of interest for I*EARN and not for the organisations employing them.

Given the scale of that organisation, this kind of work is only possible with the use of computer networks and applications such as electronic mail, computer conferencing, world-wide Web documents and videoconference.
During two years, at UPC we have been giving support for projects run from a group of Catalan schools participating in I*EARN international projects. In one of them, the "International Year of the Family", people from local schools run the project and an estimation of around 30,000 pupils from 15 countries participated in the experience. This complex and large network environment was a challenge for us, and we believe that some of the ideas of Aleph may be useful for the I*EARN computational environment. A report on the observation of their activities is described in (COMIC Deliverable 1.3), and (Serra 1995).

7.1. The I*EARN computational environment: Conference Projects

The computational environment is based in two kinds of computing services: electronic mail and electronic conferences. These two basic services are the mechanisms to carry out most of the work. Electronic mail is used to do informal communication, knowing each other, and keeping people aware of each other's work. I*EARN uses conferences for organised activities. The roles are: proj_participant, anyone involved in the project work; student, produces new items of information as a result of activities at the school; teacher, represents and coordinates a group of pupils; facilitator, responsible for the organisation of contents and filtering of one conference/project; manager, responsible for the creation and management of one conference/project.

There are basically three kinds of conference/projects: open, moderated and private. Most of the projects are open: participation is encouraged and the facilitator(s) try to keep the project running motivating participants and raising awareness on the progress of the various activities in the project. Some projects are moderated: here, the facilitator(s) is the only allowed to perform updates on the content of the conference. A few projects are closed: they are used for the coordination of I*EARN itself or to the coordination of complex projects. For most projects, the substitute of a closed conference with a small number of participants is a mailing list.

Conference items (notes) are replicated at several sites, so each participant follows the conference from one of the sites because all contributions generated at any site are sent to all the other sites as soon as possible. While this is the normal case, sometimes it is useful to send a note to just one or several sites: for instance, one note to all nodes in Argentina.

In addition, some participants with some responsibilities in the project use a special gateway that sends new contributions to a conference as personal email messages. This has proven to be helpful as an awareness mechanism for some busy users or for facilitators, because they receive new contributions as messages in their mailbox, without having to visit the conference.

From the I*EARN WWW server (http://igc.apc.org/iearn) here is an extract of how conferences are structured. At I*EARN, conferences in addition to the gopher server (gopher://gopher.iearn.org) and the various regional WWW servers
constitute the computational representation of the organisational context. There are three groups of conferences:

1. **I*EARN Conferences for projects.** The *iearn.ideas* conference is the forum to propose, approve and allocate participants in new projects. *iearn.projects* is the conference to keep a history of completed projects. Each one of the remaining conferences are devoted to one active project (projects usually last for several years, reactivated when new schools come in).

![I*EARN Project Conferences](image)

- *iearn.ideas* (Go here for project ideas)
- *iearn.projects* (Go here for reports on completed projects)
- *iearn.environ* (International Water Monitoring Project)
- *iearn.family* (Cross-cultural Family Project)
- *iearn.fp* (First/Indigenous Peoples)
- *iearn.heroes* (Cross-cultural Heroes)
- *iearn.hyp* (The Holocaust/Genocide Project)
- *iearn.kidscan* (Kids Can Elementary/Middle Newsletter)
- *iearn.math* (The Power of Math)
- *iearn.newplace* (Making Schools Better Places)
- *iearn.oneday* (One Day in the Life Cross-cultural Comparison)
- *iearn.pump* (Clean Water for Nicaragua)
- *iearn.recovery* (Recovery/Substance Abuse)
- *iearn.tc* (The Contemporary Global News magazine)
- *iearn.tolerance* (Global “Beyond Tolerance” Project)
- *iearn.wv* (Ultraviolet Radiation Ozone Depletion)
- *iearn.violence* (Violence By and Against Youth)
- *iearn.vision* (The Vision Literary Anthology)
- *iearn.youthcan* (Youth Can Environmental Action Project)

2. **I*EARN Conferences for linking people and resources.** These have the function of forums of informal non-focused communication for specific groups of people depending on their role (teachers, youth), location (mich, pnw, wasocst) and others.
I*EARN Conferences for Linking People and Resources

- learn.teachers (Faculty lounge - all teachers welcome!)
- learn.youth (Meeting place for students & young people)
- learn.practise (Practice posting messages to a conference)
- learn.sharenet (Resources for individualized learning)
- learn.options (Discussions about alternative education)
- learn.mich (Meeting place for Michigan teachers)
- learn.pnw (Meeting place for Pacific Northwest teachers)
- learn.wascocst (Washington State Social Studies Tech. Group)

I*EARN Conferences for Management and Training

- learn.trainers (Ideas and templates for teacher training)
- learn.mentors (Open to I*EARN subject area mentors)
- learn.management (Open to the I*EARN management assembly)
- learn.exec (Open to the I*EARN executive council)
- learn.fiscal (Financial planning by management assembly)
- learn.uucp (Technical discussion of UUCP systems)

7.2. A description of I*EARN in the C-MOI notation

In addition, at I*EARN regional face-to-face meetings a video-telephone is used to give a vivid realism and presence to the connections and lets pupils talk and see other people around the world working in the same project. Finally they celebrate an international plenary meeting once a year. In that meeting the worldwide I*EARN community meets all together, review last year experiences and prepare new projects.

When a teacher of the organisation wants to launch a project, first he communicates the idea through e-mail to different people in the net, and after some positive feedback, he formally request for a new project (and hence a conference) in the iearn.ideas conference with a prescribed structure. After some final discussions, articulation of responsibilities and commitments from actors to fulfil the roles required for the task (facilitators, teachers, manager(s)), a conference is created and activity starts. Here is an example of an accepted project proposal for the I*EARN TOLERANCE project (reproduced as it appears on the iearn.tolerance and iearn.ideas).
### PROJECT TOLERANCE

1. **Purpose:** 1995 has been declared by the United Nations as the International Year of Tolerance. This conference defines tolerance as the recognition and respect for the beliefs and practices of others. A number of activities will be proposed to help students get to know and appreciate people of the many different cultures and social groups on our planet. These activities will also lead students to understand and analyze situations in their own lives and in communities where intolerance exists. Students will be encouraged to take action to reverse such attitudes.

2. **Ages of the Participants:** All

3. **Subject Areas:** The project can be integrated into the curriculum in a variety of areas including: language arts, foreign languages, social sciences, artistic expression, technology, etc.

4. **Languages:** English, Spanish, Catalan, French, and other languages represented in the I*EARN network.

5. **Start Date:** June, 1995

6. **Finish Date:** We will complete the first round of activities by Nov. 30, 1995 but hope to continue this project as long as there is interest.

7. **Impact:** The project will promote: greater understanding of other people, social groups and cultures; respect for minority cultures; and the collaborative resolution of problems people experience in getting along with one another in our society. This project will serve as a point of departure for subsequent or simultaneous participation in the conference Alternatives to Violence (iearn.violence). Our goal is to develop plans for action so that our work goes beyond simply discussing these issues.

8. **Product:** A multilingual publication will be created with the most interesting contributions. Selected texts and artwork will also be placed on the WWW.

9. **Setting:** The work for this project can be done by individuals, in small groups, or as a class.

10. **Number of Participants:** No limit. The more diverse the participation, the more interesting the project will be.

11. **Method of Communication:** I*EARN, Orillas, Enlaces, Pangea, Freinet.

12. **Coordination of the project:**
   - Narcis Vives - nvives@pangea.upc.es (In Catalan)
   - Paula Perez - paula.perez@proedu.edu.ar (In Spanish)
   - Marcela Urcola - marcela.urcola@proedu.edu.ar (In English)
   - Alicia Young - alicia.young@proedu.edu.ar (In French)
   - Anna Pinyero - annap@pangea.upc.es (Global art project)
   - Antonio Cara Ribas - acara@pangea.upc.es (Clowns without borders)
   - Elena Noguera - enoguera@pangea.upc.es (University level)

In terms of the CMOI notation, the above structure can be matched to the specification of a OAW task.

**Task =**

<table>
<thead>
<tr>
<th>ATTRIBUTE-NAME</th>
<th>ATTRIBUTE-TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>1. Purpose</td>
</tr>
<tr>
<td>preconditions</td>
<td>5. Start Date</td>
</tr>
<tr>
<td>priority</td>
<td>--</td>
</tr>
<tr>
<td>postconditions</td>
<td>6. Finish Date</td>
</tr>
<tr>
<td>in-triggers</td>
<td>5. Start Date</td>
</tr>
<tr>
<td>responsible</td>
<td>proposer (<a href="mailto:nvives@pangea.upc.es">nvives@pangea.upc.es</a>)</td>
</tr>
<tr>
<td>supervised by</td>
<td>suport</td>
</tr>
<tr>
<td>has allocated</td>
<td>news:iearn.tolerance</td>
</tr>
<tr>
<td>realised by</td>
<td>Creating our own affective dictionary; Global Art Project; Reflections and analysis; Forum for debate; Action steps; Poems, songs, stories; University debate</td>
</tr>
<tr>
<td>criteria of accomplishment</td>
<td>coordination (12); policy (6)</td>
</tr>
<tr>
<td>assessed by</td>
<td>coordination (12); policy (6)</td>
</tr>
<tr>
<td>approved by</td>
<td>coordination (12); policy (6)</td>
</tr>
<tr>
<td>assigned by</td>
<td>management council, executive committee</td>
</tr>
<tr>
<td>defined by</td>
<td>admin; instructed by executive committee</td>
</tr>
<tr>
<td>adapted by</td>
<td>coordination (12)</td>
</tr>
<tr>
<td>awareness</td>
<td>--</td>
</tr>
</tbody>
</table>
In addition, in some complex projects such as IEARN.TOLERANCE, several subactivities may be defined. Here follows the description of one of such activities. One is the following:

**CREATING OUR OWN AFFECTIVE DICTIONARY**

1.-Description:
This dictionary, to be created by the students participating in the tolerance conference, will feature words that are key to our perceptions of other people but that have many different meanings in our society. The richness of its contents will reflect the diversity of the students from different backgrounds and cultures who contribute. "Words mean different things depending on who's speaking and who's listening and what we think we're talking about. They're especially unreliable when we're talking about feelings.... When people stick to their own different meanings of the same words, the stage is set for contention." (Teaching Tolerance Magazine)

2.-Proposed words to define:
- Friendship - Racism - Conflict - Religion - Politics - Country - Peace
- Well-being - Poverty - Tolerance - Love - Hate - Respect - Prejudice

3.-Suggested activities:
Step 1: Students write what each word means to them. Young students might instead want to create an illustration.

Step 2: Students ask their parents for their definitions of each of these words.

In terms of the CMOI notation, the above structure can be matched to the specification of a OAW PerfAction.

<table>
<thead>
<tr>
<th>ATTRIBUTE NAME</th>
<th>ATTRIBUTE TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>CREATING OUR OWN AFFECTIVE DICTIONARY</td>
</tr>
<tr>
<td>Content</td>
<td>1.-Description</td>
</tr>
<tr>
<td>Realises</td>
<td>(task) PROJECT TOLERANCE</td>
</tr>
<tr>
<td>initiated by</td>
<td>coordination</td>
</tr>
<tr>
<td>done by</td>
<td>pupils, teachers; at many schools ...</td>
</tr>
<tr>
<td>in, out, used</td>
<td>news:iearn.tolerance (the group of notes ...)</td>
</tr>
<tr>
<td>in-triggers</td>
<td></td>
</tr>
<tr>
<td>precondition</td>
<td></td>
</tr>
<tr>
<td>priority</td>
<td></td>
</tr>
<tr>
<td>postcondition</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>state-value</td>
</tr>
<tr>
<td>temporal constraints</td>
<td>end of the project (6.-Finish Date)</td>
</tr>
<tr>
<td>duration</td>
<td>few months</td>
</tr>
</tbody>
</table>
It is interesting to note that this is just one project (iearn.tolerance), with many proposals for activities in this task. One additional source of complexity comes from the diversity of people involved in the task, the slight differences between each news store, but specially by the use of multiple languages. In previous projects, multiple languages were managed with the recommendation to submit each note in as many languages as possible. In addition there were volunteers who completed the work. This imposes an interesting problem of awareness (translator-producer), and of synchronization between work in different linguistic communities. Perhaps additional computational artefacts would be useful for that. Here is how the iearn.tolerance looks like, just a week after the starting date.
The most important artefact used in I*EARN projects are conferences. Here is a description of the Conference Active Artefact for Open, Closed and Moderated conferences.

<table>
<thead>
<tr>
<th>ATTRIBUTE NAME</th>
<th>ATTRIBUTE TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>OPEN</td>
</tr>
<tr>
<td>access rights</td>
<td>(Open) (Any) (update, read)</td>
</tr>
</tbody>
</table>
The two basic information resources are electronic mail messages and news (notes). Here's one note as it is presented to participants in a conference:

---

**Subject:** Introduction / Introducción / Introducción / Introduction  
**Date:** Thu, 1 Jun 1995 06:48:11 GMT  
**From:** nyves@citel.upc.es (Harkis Vives)  
**Organization:** Com/Coop (Pangea)  
**Newsgroups:** learn.tolerance

1995 ANYO INTERNACIONAL PARA LA TOLERANCIA.  
1995 INTERNATIONAL YEAR OF TOLERANCE  
1995 ANY INTERNACIONAL PER A LA TOLERANCIA.  
1995 ANNEE INTERNATIONALE DE LA TOLERANCE  
TITULO: MAS ALLA DE LA TOLERANCIA.  
TITLE: BEYOND TOLERANCE  
TITOL: CAP A LA TOLERANCIA.  
TITRE: AU DELA DE LA TOLERANCE

INTRODUCTION  
The United Nations has declared 1995 as the Year of Tolerance, "helping children and young people grow in a climate of openness toward other people, their cultures and teaching them that it is important to reject violence ..."
---

In terms of the CMOI notation, the above structure can be matched to the specification of a OAW InfoRes.

**InfoRes =**

<table>
<thead>
<tr>
<th>ATTRIBUTE NAME</th>
<th>ATTRIBUTE TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td><a href="mailto:199506010648.1AA18875@citel.upc.es">199506010648.1AA18875@citel.upc.es</a></td>
</tr>
<tr>
<td>Type</td>
<td>document (note)</td>
</tr>
<tr>
<td>Description</td>
<td>Subject: Introduction ...</td>
</tr>
</tbody>
</table>
Finally, the Computation Coordination Mechanism is outlined in the following table. It describes a mechanism for I*EARN.Tolerance that uses an AA (an Open Conference), with a few procedures (proctor) where OAW are interrelated.

<table>
<thead>
<tr>
<th>Managed by</th>
<th>conference_manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocated to</td>
<td>(task) PROJECT TOLERANCE</td>
</tr>
<tr>
<td>Access rights</td>
<td>read, any i*earn subscriber</td>
</tr>
<tr>
<td></td>
<td>remove, author</td>
</tr>
<tr>
<td></td>
<td>write, facilitator</td>
</tr>
<tr>
<td>Location</td>
<td>news://citel.upc.es/iearn.tolerance</td>
</tr>
<tr>
<td>Relation</td>
<td><a href="mailto:199506011125.NAA20835@citel.upc.es">199506011125.NAA20835@citel.upc.es</a>, part relation</td>
</tr>
<tr>
<td>State</td>
<td>state-value</td>
</tr>
<tr>
<td>Governed by</td>
<td>Sender: <a href="mailto:nvives@pangea.upc.es">nvives@pangea.upc.es</a></td>
</tr>
<tr>
<td>Adapted by</td>
<td>Sender: <a href="mailto:nvives@pangea.upc.es">nvives@pangea.upc.es</a></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Finally, the Computation Coordination Mechanism is outlined in the following table. It describes a mechanism for I*EARN.Tolerance that uses an AA (an Open Conference), with a few procedures (proctor) where OAW are interrelated.

\[
C2M = \begin{array}{|c|c|}
\hline
\text{ATTRIBUTE NAME} & \text{ATTRIBUTE TYPE} \\
\text{name} & C2M-I*EARN.Tolerance \\
\text{Active Artefact} & AA (OPEN CONFERENCE) \\
\text{Procedures} & reply, post, follow-up, article, thread, catch-up, cancel, ... \\
\hline
\end{array}
\]

From the above definitions, the structure of the iearn.tolerance project may be described by the following figure.

Each AA located at different places is almost a replica of each other, but in reality, there are local interactions in one AA that are not propagated to other replicas. In addition, there are some differences between each AA due to the use of conferencing systems with different functionality. Their relationship may be described by means of over-write primitives of the interoperability language (i.e. the over-rule mode of relation).

After one month of work, one project coordinator has requested for the reorganisation of notes in the conference because some contributions were not properly submitted, and the number of participants made the conference difficult to follow. That was not predicted in the design of the system, and if implemented, it would require to modify every note: the conference_manager (note.Managed_by), the sender (note.Governed_by), to add facilitator in
note. Adapted by, so that the facilitator is enabled to reorganize the relationships between notes (several attributes in the note: subject:, followup-to:, in-reply-to: ...). The cooperative modification in all the AA will be once more done by an over-rule relation.

8. References

COMIC (Computer-based Mechanisms of Interaction in Cooperative Work) CEE, ESPRIT Basic Research Project 6225.
Chapter 4

Let’s Talk About Bugs!

Towards Computer Support of the Coordination of Software Testing

Peter H. Carstensen, Carsten Sørensen
Systems Analysis Department
Risø National Laboratory

Tuomo Tuikka
Department of Information Processing Science
University of Oulu

Abstract: Software testing is often a complex process potentially involving a large number of geographically distributed people with different perspectives and competencies. Software testers, software developers and project managers engage in discussions about the software errors found, they negotiate the relative importance of the bugs, they allocate responsibilities and resources, they coordinate who is doing what, etc. They talk about bugs. In order to coordinate and manage talking about bugs, a number of means for coordination are applied. The aim of this paper is to analyze coordination work in software testing in order to promote general requirements for computer support. We have studied the testing of more than 200,000 lines of code at Foss Electric, a Danish manufacturing company, and focused on two aspects: Firstly, the coordination activities related to the process of distributed registration, classification, diagnosis, correction, and verification of software errors, as well as the monitoring of the state-of-affairs of testing activities. Secondly, the mechanisms used to support the coordination. The analysis resulted in the identification of the need for computer support for coordinating this part of the software testing process, e.g., support of distributed classification, routing of information, and facilities providing an overview of state of affairs.

1. Introduction

“The lay public, familiar with only a few incidents of software failure, may regard them as exceptions caused by inept programmers. Those of us who are software professionals know better: the most competent programmers in the world cannot avoid such problems. [...] Software is released for use, not when it is known to be correct, but when the rate of discovering new errors slow down to one that management considers acceptable. [...] It is not unusual for software modifications to be made in the field. Programmers are transported by helicopter to Navy ships: debugging notes can be found on the walls of trucks carrying computers that were used
in Vietnam. It is only through such modifications that software becomes reliable.” (Parnas, 1985)

Software testing is an extremely complicated activity. In practice an exhaustive test is impossible (Myers, 1979; Parnas, 1985). Despite of all the techniques and methodologies for specific source code testing, black box testing, usability testing, etc., no methodologies exists for establishing a set of unambiguous criteria for a sufficient test strategy in order to ensure that the product is reliable, usable, and correct (Petchenik, 1985). Hence, much effort is required to establish a common understanding among software developers, software testers, and software managers of when a product is acceptable. Organizations involved in software testing typically apply the strategy of having people with different skills and perspectives test the software. As argued by Dahlbom & Mathiassen (1993): “Effective quality control requires a certain division of labor and responsibilities. In practice, quality is not the only concern, and there is a constant struggle between quality and resource interests. Independence is needed to constantly defend a quality position and to avoid the self-deception in having systems developers evaluate their own products.” The division of labor in the software testing process can either be done so that different actors perform different subtasks (detection, diagnosis, correction, etc.), or it can be organized so that each person has the responsibility for all testing activities within a limited part of the program. In any case, the participants will inevitably be mutually interdependent. As argued by Parnas (1985), the need for coordination in software development can not be eliminated by structuring the software properly. In order to mesh their work results, interdependent actors performing distributed software testing tasks must coordinate and negotiate their work (Schmidt, 1994; Kraut and Streeter, 1995).

Recent works have addressed cooperative aspects of software development. Johnson & Tjahjono (1993) promote the CSRS system supporting collaborative software review, and Mashayekhi et al. (1993) describe the CSI system supporting distributed collaborative software inspection. Swenson et al. (1994) show by example how a workflow system can support the coordination of distributed software testing. Kraut & Streeter (1995) present an analysis of formal and informal aspects of coordination in the software development process, focusing on the coordination conducted by peer-to-peer communication, by project schedules, and by review and inspection meetings. They argue for the importance of informal direct communication in systems development, but at the same time argue that the excessive transaction costs and the ephemeral nature of the information transferred in informal communication implies that more formal coordination means must be applied. We focus is on how formal means can support coordination of distributed software testing activities.

We have studied coordination work in the testing of more than 200,000 lines of code at the Danish manufacturing company Foss Electric. Our analysis focused on the aspects of coordination supported, stipulated and mediated by various paper- and computer-based coordination tools, e.g., a paper-based bug handling
Let's Talk About Bugs!

A workflow system, a centralized binder containing bug forms, a software module integration procedure, and a project resource schedule. More specifically, we have analyzed how these tools supported the coordination of distributed registration, classification, diagnosis, correction and verification of software errors. By stipulating who is doing what, the tools provide persistent accounts to support the more ephemeral coordination work. This paper does no present an analysis of the of software testing in general.

Based on the analysis, we have identified needs for computer support for this part of the software testing process, e.g., support of distributed bug classification, routing of information, monitoring of state of affairs, etc. The purpose of applying such computer based coordination tools is by no means to remove the need for peer-to-peer communication and review meetings. It is rather to provide means of coordinating a multitude of detailed decisions which can form the basis for talking about bugs.

The next section describes the research approach applied. Section 3 gives an overview of the field of software testing. Section 4 presents the Foss Electric case. Section 5 discusses the need for computer support of the articulation of distributed registration, classification, diagnosis, correction, verification of software errors, and for monitoring the state of affairs of the testing process. Section 6 discusses our results.

2. Research Approach

This paper is based on data collected in an empirical study of one development effort at Foss Electric. The study focused both on the coordination of software testing and on the coordination of engineering design and process planning. This paper only reports on the coordination of software testing. The field study and the preliminary data analysis lasted a total of six months and was exclusively based on qualitative data collection techniques such as qualitative interviews (Patton, 1980), observations, study of project documentation, and participation in project meetings. A total of 21 interviews were conducted, and we participated in 10 project meetings. Approximately 75 man-hours were spent observing the development process. The approach was inspired by perspectives promoted in several research efforts (cf. (Bucciarelli, 1984; Schmidt and Carstensen, 1990). As argued by Siemieniuch (1992), field studies are important in order to obtain a coherent understanding of how computer tools can support product development in a manufacturing setting. The data analysis was based on theories and conceptualizations from the field of Computer Supported Cooperative Work (CSCW), as promoted in Schmidt (1994).

Yin (1989) distinguishes between a case study approach and an ethnographic approach. The former being structured and targeted, and the latter being more unstructured and primarily based on observation. Our approach can be characterized as having elements from both types with a predominant case study
bias. Although we did not start out with a strict set of hypotheses, we did bring an articulated perspective. The purpose of the study was to investigate how various paper-, board- and computer-based mechanisms supported the coordination of distributed work (Sørensen et al., 1994; Carstensen et al., 1995).

A qualitative approach offers the obvious strength of providing rich and detailed data, enabling a deep understanding of the conditions under which work is performed. It does, however, present a major limitation in terms of promoting statements of general validity. As Mason (1989) argues, the purpose of research must be to provide both the richness of detail and relevance of research problems studied, as well as a certain tightness of control or rigor. We do not believe that one empirical effort necessarily needs to encompass both aspects. We do, however, recognize that since the results reported in this paper are drawn from a single field study, we can neither make claims as to the generality of the findings, nor to a rigorous research approach. The organizational culture at Foss Electric favors both individual and group achievements, work is primarily organized in projects and it is not a particularly hierarchical organization. Various coordination systems were both designed and used by project members without leading to conflicts or fear of being monitored. It is, therefore, reasonable to assume that the types of phenomena studied at Foss Electric can only be made subject to generalization if an organizational culture of a similar nature is observed.

3. Software Testing and its Coordination

“That was back on Mark I. It was in 1945. We were building Mark II—and Mark II stopped. We finally located the failing relay and, inside the relay, beaten to death by the relay contact, was a moth about three inches long. So the operator got a pair of tweezers and carefully fished the bug out of the relay and put it in the log book. He put scotch tape over it and wrote, “First actual bug found.” And the bug is still in the log book under the scotch tape and it is in the museum of the Naval Surface Weapons Center at Dahlgren, Virginia.” Grace Murray Hopper quoted in Jennings (1990).

“The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it is a disguise that the error is the programmer’s own creation.” (Dijkstra, 1989).

The understanding of bugs in programs has changed since the 1940’s, although the idea of having an error in a program is as unpleasant as having a bug inside the computer. The metaphor describing a software error as a bug is confusing. An error can cause reactions as if it was a living insect that should be removed by using poison while programming, but it is, of course, created by the programmer and should as such be regarded as a software error. The word bug is stuck in our language and it is probably as difficult to get rid of as errors in software. We will call them bugs since our focus is more in talking about them, than on studying formal techniques for finding them.

The art devoted to finding software errors is called software testing. Myers (1979), for example, defines software testing simply as being the process of
executing a program with the intent of finding errors. The view to software testing has been changing during the last decades. The early view of programming and testing was that you “wrote” a program and then you “checked it out.” Later testing has been defined as evaluation of software or prevention of problems. An illustration of the evolution in software testing can be found in Gelperin and Hetzel (1988). Hetzel (1988) for example defines software testing as any activity aimed at evaluating an attribute or capability of a system, and determining that it meets its required results. It is widely accepted that software testing is one of the corner stones of modern software engineering and thus should be an integrated part of any quality program, and a central activity in all quality assurance groups (Yourdon, 1988).

The field of software testing spans mathematical theory, the art and practice of validation, and methodology of software development (Hamlet, 1988). Software testing literature mostly addresses the relationship between the testing and the software engineering process (i.e., the use of testing methods and tools). The cooperative aspects of the process is only marginally addressed. It is important to consider software testing as a part of the software engineering process, i.e., relate the stages in the process to stages of testing, e.g., unit, integration, system, product, customer, and regression testing (Dalal et al., 1993). A broad array of testing methods and techniques are available today, e.g., black and white box testing techniques providing a systematic approach to the design of test cases (see for example (Beizer, 1990) and (Hetzel, 1988)). A fast growing flux of automated software testing tool products influences the field today. Neither the stages, the techniques, or the automated tools will, however, be discussed further in this paper.

The vast majority of software organizations have substantial room to improve the manner in which testing is conducted and software testing is often the poorest scheduled part of programming (Brooks Jr., 1982; Pressman, 1988). If the importance is not recognized correctly, the project planning will not include enough time. This causes many problems to the testers, i.e., accumulated pressure in the work. The complexity of the testing work and a tight schedule influence the decisions determining whether or not a software product meets its requirements. That is, there is no systematic way to search, no way to judge points selected, and no way to decide when to stop (Hamlet, 1988).

Since exhaustive testing is impossible (Myers, 1979; Parnas, 1985), a common understanding of the status of a software product can only be established by means of negotiations. These situations are not easy to cope with and will often result in work settings including many cooperating actors. The actors in an ensemble developing and testing software become interdependent: “Cooperative work occurs when multiple actors are required to do the work and therefore are mutually dependent in their work.” (Schmidt, 1991). In order to handle the underlying interdependencies among the cooperating and mutually interdependent actors, a set of “second order activities” is needed. The actors must, in often complex ways, coordinate the plurality of tasks and relationships between actors.
and tasks. We use the term coordination main as the direction of individuals efforts towards achieving commonly and explicitly recognized goals and “the integration or linking together of different parts of an organization to accomplish a collective set of tasks” (Kraut and Streeter, 1995). Notice, that by this definition testers, designers, etc. do not necessarily share goals (Bannon, 1993). In our terminology, coordination can include activities aimed at negotiating, establishing, maintaining, and refining the conceptual structures and salient dimensions along which the coordination must be conducted (Schmidt, 1994).

4. The Foss Electric Case

“With the number of developers involved, it is extremely important that all problems are registered, otherwise they just ‘disappear’ [...] An important derived product then, is a list of problems reported fixed but not yet tested. Based on the lists and the problem descriptions, the platform master can check and then report the problem corrected.” (Software designer at Foss Electric)

Foss Electric is the largest manufacturer of highly specialized equipment for analytically measuring quality parameters of agricultural products in the World. The company is localized in Denmark with service and distribution offices in many countries. They employ approximately 700 people. The customers are laboratories, slaughterhouses, dairies, etc. Foss Electric is a matrix organization, and development of new instruments is organized in projects which typically include specialists with design competence in the fields of mechanical-, electronic- and software design, as well as in optics and chemistry.

4.1 The S4000 Project

The objective of the S4000 project was to build an instrument for analytical testing of raw milk. It included functionality which previously had been placed in several instruments, and furthermore introduced measurement of new quality parameters of milk. 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 S4000 was the first product featuring a built-in an Intel-based 486 PC. Configuration and operation of the instrument is done through 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. At most 50 people at a time were directly involved in developing the instrument, and the project lasted approximately 2 1/2 years. The core personnel involved in the design included a number of designers from each of the areas of mechanical design, electronic design, software design, and chemistry. In addition there was a handful of draught-persons and several persons from each of the departments of production, model shop, marketing, quality assurance, quality control, service, and top management. A group of between 5 and 12 software
designers was involved in designing and coding the software required to operate and control the S4000.

4.2 Coordinating the Bug Handling Activities

During the S4000 project, the software designers realized problems in coordinating, controlling and monitoring the software testing activities. This and external requirements for more precise measurements of the status of the software testing process led to a standardized bug form and a centralized binder being invented, used, and refined by the involved software designers during the S4000 project for registering and filing identified bugs. Furthermore, a set of concomitant procedures and conventions for the use of the form and binder were established. Some of these were formulated by the designers themselves and written down as organizational procedures, others were established as conventions agreed upon by the software designers. The bug form and the general procedure for using the form are illustrated in Figure 1. The purpose of the form and the procedures were to ensure that all identified bugs were registered and “remembered” until they were corrected. This was accomplished by ensuring that each registered bug was represented by one form only, and by ensuring that changes to the state of the process dealing with a bug was reflected in the form representing the bug. To ensure that all bugs were handled, a central file (the binder) contained a copy of the form until a final state was reached, and the original form was filed. Several groups of actors and roles were involved in this process, i.e., users of the bug form, the binder, and the procedures. These were:

- Testers from different departments and with different perspectives on software quality involved in testing the S4000 software. Apart from the software designers approximately 20 other actors were involved in testing the software.
- The spec-team, a group of three software designers responsible for diagnosing bugs and deciding how to handle the correction of bugs. These persons represented different areas of expertise in relation to the software architecture.
- Software designers each responsible for one or more software modules.
- The central file manager who was one of the software designers responsible for maintaining a binder containing forms for all registered bugs.
- The platform master responsible for managing and coordinating the activities in one of the integration periods (called a platform period).
- The plan-manager responsible for updating the work plans. In the S4000 project the plan-manager was one of the spec-team members.
| Description: | (3) |
| Classification: | 1) Catastrophic  
2) Essential  
3) Cosmetic | (4) |
| Involved modules: | |
| Responsible designer: | |
| Estimated time: | |
| Date of change: | |
| Time spend: | |
| Tested date: | |
| Periodic error - presumed corrected: | |
| Accepted by: | |
| Date: | |
| To be: | 1) Rejected  
2) Postponed  
3) Accepted | |
| Software classification (1-5): | ___ | |
| Platform: | |
| Description of corrections: | (8) |
| Modified applications: | |
| Modified files: | |

The actors fill (or add information) in:
- The testers: (1), (2), (3), and (4)
- The Spec-team: (3), (4), (5), and (7)
- The designers: (6) and (8)

The procedure for handling bugs:
- A tester register and classifies a bug (field 1, 2, 3, and 4)
- The tester sends the form to the spec-team (field 3, 4 and 7)
- The spec-team diagnose and classify the bug (field 3, 4 and 7)
- The spec-team identifies the responsible designer (field 5)
- The spec-team estimates the correction time (field 5)
- The spec-team incorporates the correction work in the work plans
  - The spec-team requests the designer to correct the problem (field 5)
  - The designer corrects the bug and fills in additional correction information (field 6 and 8)
  - The designer sends the form to the central file (field 6 and 8)
  - The CFM sends the form to the central file manager
  - The PM verifies the correction (field 6 and 8)
  - The PM returns the form to the central file manager

The routing of the bug forms among the six roles were done according to the following eight steps (see Figure 2):
1. A tester sends a form to the spec-team describing a registered and classified bug
2. The spec-team adds diagnosis and estimation information and sends it to a software designer
3. The spec-team requests the designer responsible for the plans to update the planning spread-sheet
4. A copy is sent to the central file manager. If a bug is rejected the original is sent to the central file manager
5. The software designers add correction information to the form and send it to the central file manager
6. The central file manager sends a pile of forms to be verified to the platform master
7. Forms which can not be verified are send to the spec-team
8. Verified forms are send to the central file manager
All registered bugs were filed in a binder providing all software designers and other testers with access to the state of affairs in the testing process. During one and a half year approximately 1400 bugs were registered, treated and filed in the S4000 project. The binder was physically placed in the room use by the project team (all the designers, but not necessarily the testers). The binder had the following seven entries reflecting the status of a specific bug, and in each of these entries the forms were filed in chronological order:

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)
7. Corrected bugs (originals).

The bugs reported in the S4000 project were handled according to the twelve procedural steps listed in Figure 1. In most cases the prescribed procedures were followed strictly. There were, of course, situations in which the actors did not follow the procedure. A thorough step by step description of the procedure and of the “typical” exceptions is given in Carstensen (1994).

An interesting and important characteristic of the software development and testing work in the S4000 project was the organization of software development and the structuring of plans in working cycles called “platform periods”. A platform period was typically 3–6 weeks of development followed by one week of integration. Version 1 of the S4000 system covered approximately 15 platform periods. As a configuration control measure, revisions of the software were only allowed after the “platform” had been released. For each platform period, a platform master was appointed by the group of software designers. The platform master was responsible for collecting all information on updates and changes made to the software, for ensuring that the software was tested and corrected, and
for ensuring that the project schedule was updated with revised plans and activities before the platform was released.

In the S4000 project one of the spec-team members was responsible for the overall plans. He was responsible for incorporating the new tasks (e.g., correction tasks), changes, etc. into the plans. The plans were organized in a large spreadsheet containing information on: tasks to be accomplished and references to detailed descriptions of tasks, estimated amount of labor-time per module for each task, responsibility relationships between software modules and software designers, relationships between tasks and platform periods, and total planned work hours per platform period for each software designer.

In order to coordinate the activities of handling bugs in the project a multitude of discussions, ad hoc meetings, and planned meetings were conducted throughout the project. In relation to the registration of a bug, the testers engaged in discussions of the problems they had identified, or they discussed the classification of a bug with one of the designers. The process of diagnosing bugs was organized as a structured meeting once a week where the spec-team (and sometimes other involved designers) discussed and negotiated the diagnosis and classification of the reported bugs, and how to incorporate the correction into the plans. Based on our observations, one out of four bug reports required discussion and negotiation between a tester and a designer, or between the spec-team and a designer. A spec-team member typically spent one day a week diagnosing bugs and negotiating bug classification and resource allocation with testers and software designers. When correcting bugs the designers often engaged in ad hoc discussions with other designers and in negotiations with, for example, the marketing people (who were responsible for the overall requirements) of the actions that would be acceptable. Verification was done during the integration periods. The designers spent much time during the integration periods negotiating acceptable solutions to the problems that had occurred. The integration process also contained a structured meeting in which all software designers participated. At this meeting all problems and solutions were presented. Our observations indicate that during integration, all of the software designers spent half their time coordinating the software integration and negotiating how to deal with the problems at hand.

As outlined above, coordination activities were also supported by means of forms, lists, procedures, etc.: (1) The bug form and the related procedure stipulated the flow of the bug registration and correction work; (2) the spreadsheet provided a conceptual structure for scheduling tasks, actors, and deadlines by relating development activities to relevant software modules and to responsible actors; and (3) the platform periods established a common basis for the designers’ activities, thus facilitating overview of the state of affairs, guaranteeing that the software components could be integrated, and that the corrections conducted were verified. Figure 3 provides an overview of the artifacts involved in the coordination of software testing. The invention, implementation, refinement, and use of new means supporting the coordination evolved fast and without serious
complications. A small group of people organized the work and several of the forms, lists, etc. were invented due to needs realized by the group itself.

The characteristics described above are in many respects similar to what can be observed in office work, e.g., the activities and roles (Hirschheim, 1985), although they mainly address the coordination aspects of the bug registration and correction process. The complexity of the actual cooperative work, e.g., the problems of identifying the cause of a bug, will not be addressed in this paper.

![Diagram of bug handling process]

Figure 3: Artifacts and prescriptions supporting the coordination of the bug handling process. The registered bugs create new tasks which are incorporated in the project plan. The project plan defines when the next platform integration must be started. The bug forms provide input for the platform integration by specifying verification tasks. The binder and the procedure for handling bug forms ensure that all bugs are treated, and in addition they provide an overview of state of affairs in the process.

5. Identifying Needs for Coordination Support

The software testing process in the S4000 project at Foss Electric contained a large number of activities. In this section we present and discuss coordination work related to the activities of: registration, classification, diagnosis, correction, verification, and monitoring the state of affairs. For each of these activities we furthermore discuss the functionality supporting the coordination which could be provided by computer technology. In order to structure the discussion, these activities are presented as distinct stages of work, and for each stage the analysis of the work and the discussion of need for computer support are presented in two separate subsections. Section 5.1 discusses support of the work flow, and the following sections discuss each of the stages of work.
5.1 Supporting the Work Flow of Software Testing

**Work:** In the S4000 project, registration, classification, and correction of bugs were distributed activities. Diagnosis of software errors was done by the spec-team at a weekly meeting. This meeting resulted in correction requests being distributed to the 5–10 software designers responsible for correcting the bugs. The platform master was, during the integration period, responsible for verifying the corrections. As an example, a marketing person tested the usability of the user interface and realized that some functionality was not accessible from the menu structure. The problem was then classified and described in a bug form and sent to the spec-team. They decided that Hans, as responsible for the UI-module, should make a correction. After Hans had completed the correction he reported to Jens, who was platform master for the next integration period, that the problem had been dealt with. During the platform integration week Jens tested and accepted the corrections. In summary, this process was organized as distributed testing, centralized diagnosis, distributed correction, and centralized verification.

**Support:** A work flow of distributed error registration, classification, diagnosis, correction, and verification, among other things, needs support for routing information between actors. When an actor has completed his or her activities in relation to a specific bug form, the work flow system must automatically validate that the required information is registered, then pass the information on to the next actor (or group of actors), and finally notify the receiver(s) that new action must be taken. However, in most situations is it impossible to completely specify all situations which may occur (see e.g., (Suchman, 1987; Schmidt, 1994) . The coordination of software testing at Foss Electric was certainly no exception to this. Thus, the actor completing an activity must be able to overrule the routing and redirect the information to somebody else. The protocol stipulating the routing must furthermore be based on roles to which actors can be related. The study at Foss Electric clearly illustrates that the actors had several roles, and, more importantly, that these roles were relatively stable entities in the work setting. Some roles were played by different actors at different points in time, e.g., the role of platform master. Changing the actor related to a role should not imply changes to the protocol stipulating the routing. Monitoring the progress of the work is essential when coordinating work. This requires a consistent and updated database containing information on all reported bugs and their current status. Negotiation of classifications, diagnoses, allocation of resources, deadlines, etc. were a predominant feature of the software testing work. Approximately one out of five of the correction tasks defined by the spec-team resulted in negotiation between the spec-team and the responsible designer. Support for actors engaging in negotiating the bug handling process could draw upon research addressing the support for negotiation (Flores et al., 1988) , or available work flow technologies (e.g., (De Cindio et al., 1988) . It is beyond the scope of this paper to review this further.
5.2 Registration and Classification

**Work:** The S4000 software was tested on software simulators and on instrument prototypes, and the project had distributed detection, registration, and classification of software bugs. Occasionally testers and software designers engaged themselves in preliminary discussions on the interpretation of problems or possible bugs. Some errors were impossible to reproduce and hence difficult to describe during registration. Our observations indicate that in at least 20% of the bug reports, testers were not able to describe the problem in detail. If it was difficult for the tester to fill in the required information in the bug form, the spec-team member filled in the form after having discussed the problem with the tester. As one of the spec-team members said:

“The form is not very user-friendly. We often have to force them [the testers] to fill in a form. Sometime they just send in a note describing what they have seen, and we must produce a form.”

The classification of errors as either catastrophic, essential or cosmetic was done according to the tester’s perspective. As one of the spec-team members phrased it:

“People, depending on who they are, often interprets a catastrophe in a different way than I do. An inconsistency in the user interface might, for example, be a disaster to a marketing guy, whereas it is a cosmetic problem to me”.

Some of the testers tried to maintain awareness of errors identified by other testers in order to obtain useful information for a first diagnosis of the problem. This was, however, a difficult task. If, for example, a tester from the marketing department wanted to check recent problems with the menu structure, this could imply browsing through more than 500 forms in the binder.

**Support:** The bug handling system studied could be improved by refining the bug classification system. Too often the existing classification structure led to discussions resulting in the spec-team re-classifying bugs. A more elaborated classification of the type and importance of bugs could support the testers in providing useful information to the spec-team and to designers. We suggest two classification structures: A classification of the phenomena observed (program stopped, window in wrong place, unstable output on tests, etc.) and a two-dimensional classification structure reflecting software quality parameters (maintainability, marketing, stability, safety, usability etc.) and the testers assessment of the level of importance. Research within software engineering may provide input for more elaborate classification structures, for example, standard software quality taxonomies (Boehm, 1981; Fairley, 1985). If the classification structures have a “miscellaneous” category on each level, the software tester is provided with an opportunity to classify in the most unambiguous way. These “other” fields might contain text annotations, allowing testers to further specify and characterize the problem. Support for filling in bug forms could be supported through a facility for retrieving registrations of similar bugs based on more elaborate classification structures and using a central database containing...
information on all bugs. Classification structures reduces complexity of coordination work by providing a conceptual structure that makes it possible for testers and designers to perform distributed storing and retrieval of bug forms with a minimum of peer-to-peer coordination. Also, support for discussions via electronic mail or bulletin boards among testers could improve the quality of the information registered.

5.3 Diagnosis

Work: Diagnoses were mainly done by the spec-team members at a weekly meeting—more frequently in periods. They browsed the submitted bug forms. The number of forms varied a lot from week to week—in the last half a year of the project it was around 25–30 forms per week. The status and classification of each bug were discussed. If the classification differed very much from the one made by the tester, the team sometimes summoned the tester and discussed the classification with him. If two registered bugs were diagnosed as being identical, only one of the forms was processed further. The spec-team then discussed the diagnosis and responsibility by reviewing specifications, documentation, source code, etc. In the complicated cases (about 10%) the team summoned the designers responsible for the relevant modules in order to negotiate the diagnosis, the responsibility, and an estimated correction time. Responsibilities and time-estimates for corrections were incorporated as tasks in the planning spread-sheet. The allocation of correction tasks to platform periods was decided based on assessments of the workload of the responsible designer(s) and of the importance of the problem. The spec-team either handed over or sent bug forms to the designers. This could result in discussions among designers and spec-team members about the diagnoses and time-estimates.

Support: Supporting diagnosis of bugs primarily implies supporting communication among the spec-team members. The field study clearly illustrates, that without face-to-face communication, the spec-team members would have had severe problems. E-mail based communication between spec-team members, testers and designers might, however, support diagnoses and prevent some of the ad hoc discussions. The coordination of the diagnosis work could also have been supported by providing access to information on already reported bugs. As discussed in the previous section, improved classification structures could provide better support for diagnosing bugs by more clearly stipulating testers' assessment of type and importance of the bug. The task of meshing new correction tasks and existing plans is quite complicated. This could be supported by providing access to information on: relationships between roles and actors, architecture of the software complex, relationships between software modules and responsible designers, designers’ workloads, existing work plans, and relationships between tasks and deadlines, etc. The needs for obtaining an overview of the existing bugs and plans will be discussed further in Section 5.6.
5.4 Correction

**Work:** Bugs were corrected in a distributed manner. Although the structure of the bug form did not support the allocation of responsibility for correcting one particular bug to several designers, this often happened. Often a designer discovered that the problem he or she was correcting affected modules owned by other designers, thus creating a need for coordination of who was going to do what, when and how. Since all the 5 to 10 designers making corrections were placed in the same room, this coordination was mainly conducted by an abundance of meetings and discussions. This, however, imposed a problem when the complexity of the software increased. One of the designers characterized the problem as follows:

“The problem we have right now is that the software architecture is difficult to decompose so much that one designer can handle a component. We are all working on several components, and work on a single component involves two to four men, and perhaps even some of the electronic designers too. Then we need coordination [...] We have recently started a process where we try to produce more formal documents and agreements about the things we work with, we haven’t been good at doing this before, but now we have to do it.”

The first thing a designer usually did upon receiving a request for correcting a bug was to consider both the diagnosis, the estimated correction time and the deadline—the platform period. If the diagnosis or estimate did not seem acceptable, the designer contacted one of the spec-team members in order to negotiate the situation. With respect to the estimates, the spec-team had a policy stating that the designer was always right. When requirements could not be met within the scheduled time limit, this led to negotiations with the marketing people responsible for the requirements specification.

**Support:** The most obvious support for coordinating software correction activities is by providing good and accessible means for communication. This was to a certain extent already facilitated by placing all designers in the same room but electronic communication systems might be of help as well. Furthermore, support for the process of requesting and rejecting correction tasks must be provided, i.e., if coordination is supported by a workflow system, the designers must be able to reject a request—return it to the originator with a comment—or pass the request on to another designer pointed out to be the one that should have been assigned to the task. Improved formalization and structuring of the specification of modules, module interfaces, message handling, etc. could also support the coordination (Parnas, 1985). In the S4000 project interactions among modules were only very loosely sketched, thus resulting in designers engaging in a ad hoc coordination. Improved use of some of the existing specification techniques or CASE tools could decrease the need for ad hoc coordination by providing an improved structuring of the field of work, i.e., the software system being designed (Mathiassen and Sørensen, 1994). The designers pointed at the problem of being aware of the changes other designers made, or to be able to ensure that all the others were aware of a correction, idea, or problem. Improved support for the documentation of corrections could, for example, be provided by classification...
structures and browsing facilities for the database containing information on the bugs and corrections.

5.5 Verification

**Work:** In the platform integration process, initiated and managed by the platform master, all modules were linked and compiled, and the software was tested prior to its release, all corrections reported were verified, tasks identified during integration and testing were meshed into the plans, and designers were informed about the state of affairs. During the platform period all designers worked on testing the integrated software and were constantly meeting and discussing the results and problems of the integration. The platform master coordinated the activities and delegated subtasks to the designers. As a designer put it:

“Usually we produce a list of all the problems we have identified on a large white board. We then discuss whether this is a problem—an error—or not. Actually it’s the platform master who does that. If it’s a real heavy problem you are immediately summoned and asked to correct it.”

A set of brief organizational procedures stated how to organize the integration process, and contained a number of check lists and standards covering the details to be checked before the platform master could release the software for further modification. Only one pre-scheduled and structured meeting was held at the end of the integration period. All designers had to participate in this meeting where they in turn described the changes they had implemented since the last integration period.

**Support:** Close interaction among software designers during verification is essential. At Foss Electric this was solved by placing the designers in the same room. In cases involving a large number of project members or geographically distributed development this solution is not feasible. Here a work flow structure supporting the division of labor, a structure for establishing a common language (e.g., classification structures, module specifications, etc.) and electronic meeting and communication systems could provide support. In the coordination of concrete verification tasks, support for distributing responsibilities should be provided. In the S4000 project this was handled by the platform master who personally delegated each correction task. It would be obvious to support this by having a computer supported procedure for delegating the verification tasks, and support in reporting back. This, of course, would also include support for registering new bugs, similar to what was discussed in section 5.2. In order to improve the awareness of changes made by others and to establish a common understanding of the software complex, some support for “viewing” the structure of the software complex must be provided. From our observations it was obvious that the designers had problems in relating themselves to the structure of others designers’ modules although these had an essential impact on how they should (re-)design their own modules. This could be supported by the production technology dimension of CASE tools (Henderson and Cooprider, 1990).
5.6 Monitoring

Work: A central activity in coordinating the process of registering and correcting bugs in the S4000 project was to establish an overview of the state of affairs, i.e., the progress of the process, the number of bugs that still needed to be corrected, the accumulated estimation of correction time, changes that might affect other modules, etc. The designers needed to be aware of corrections and changes affecting their modules. The spec-team members needed to know the state of affairs before each spec-team meeting. The testers frequently tried to obtain an overview in order to avoid wasting their time on reporting already registered bugs. The platform master needed an overview of corrections to be verified in the next integration period in order to plan the integration work. Management tried to get an overview of the progress of the whole development project. There were basically three sources for this type of information: informal communication, the bug form binder, and the list of bugs which had not been corrected. There was a lot of informal communication and discussion among the designers about what kind of corrections and changes they had made. Some of the testers discussed changes in the software with the designers several times a week, whereas others never contacted the designers directly. Even though the designers sat in the same room and were engaged in discussions every day, it was difficult for them to be aware of the state of affairs:

“Usually, the channel driver guy and I had a clear deal. Verbal discussions and a sketch drawing were sufficient. But in a project as large as the S4000 we don’t have a complete overview of the software complex. And then you are in big trouble when the other guys change their code” (Software designer in the S4000 project).

Testers as well as designers found it very difficult to obtain the necessary overview by consulting the bug form binder, mainly because the forms were only organized according to the seven categories presented in Section 5.2. This made it almost impossible to determine whether the same bug had been reported in several bug forms. It was the intention that a specification of the corrections should be included in each form (cf. the bottom of the form in Figure 1). In order to be aware of relevant changes, the designers were expected to browse through all the forms in order to see if anything there was of interest. The binder contained approximately 1400 forms at the end of the project!

The third source for getting an overview was a weekly produced list of registered bugs that had not yet been dealt with. One of the designers phrased the problems with this as:

“Originally the intention was to produce statistics of the number of known-but-not-yet-fixed problems and use this as a management tool. The management hoped to find a decreasing curve on the week-to-week measurement. They didn’t. But we realized that as a management tool this can only be used if you have a stable product. We didn’t have that.”

Support: Obtaining awareness of the state of affairs by monitoring progress of the software testing process plays an important role as a fundament for coordinating activities, and this should be supported by several means. More elaborate classification structures for bugs (cf. Section 5.2), for the software
modules and their interaction (cf. Section 5.5) and for the relationships between actors, roles and resources could facilitate assessments of the relative importance of these issues. Providing designers and testers with browsing and query facilities to a database containing all registered bugs would enable these actors to access aggregated information on reported bugs which have not yet been corrected, the number of a specific type of bugs, the number of not yet corrected bugs in a specific module, the number of bugs a specific designer is responsible for getting corrected, etc. Also access to view the project schedule would be useful for monitoring state of affairs. This functionality could be provided by means of some of the existing project planning tools (e.g., Microsoft Project), and should support requests like: Who is responsible for the UIS-module? Which modules are John responsible for? How busy is Tom the next integration period? How much time has been spent on correction so far? What percentage of the corrections does this correspond to? etc.

6. Discussion

We have, in this article, analyzed the coordination of distributed software testing of one project in one organization. We have focused on coordination work related to distributed actors performing registration, classification, diagnosis, correction and verification of software bugs applying a bug form work flow system, a resource-planning spread-sheet, and a configuration control procedure. Based on this analysis we have discussed needs for computer supporting this coordination work. The project we have looked at is most likely not unique. Many software projects faced with the immense complexity of distributed software testing use various kinds of forms, classification structures and organizational procedures in order to cope with the complexity of coordinating of software testing activities.

Kraut and Streeter (1995) argue that computer support of the informal and direct communication in systems development is required, and they suggest tools supporting conferences and distributed meetings. The analysis in this paper can be viewed as work elaborating on Kraut & Streeters conclusion that informal communication is very important in systems development, but faced with the need for coordinating an abundance of distributed decisions, there is a need for formal coordination means due to the excessive transaction costs and the ephemeral nature of informal communication.

Conceptual structures played a crucial role for the actors when coordinating the software testing activities. The software was divided up into a set of modules, which also were represented in the project schedule. The architecture of the system represented an aggregation of the software modules. Work plans represented structures of actors and roles involved, resources available, tasks to be performed. Software bugs were classified according to their importance and the bug form played an important role in creating and representing bugs as separate entities, i.e., the bug form was the instrument used for transforming observed
phenomena to a set of identified software bugs. These structures provided the actors with an overview of both the field of work which was registration and correction of bugs in the S4000 software, as well as of the cooperative work arrangement, e.g., the actors, their roles, and the resources available. The structures can be viewed as dimensions along which coordination is conducted (Schmidt and Simone, 1995), i.e., the coordination activities are performed by referring to abstractions and conceptualizations of the nature of the work, not by directly interacting with the objects of the work (e.g., the code). We have identified a set of actions performed by the actors in relation to these conceptual structures. They were classifying bugs, tasks, modules, etc., routing information and requests, monitoring state of affairs, allocating resources, meshing work products, and negotiating diagnoses, allocation of resources, etc. When discussing computer support of coordination aspects of software testing, it is obvious to require access to structures in the computer based system similar to the conceptual structures mentioned above, and to facilities supporting the basic actions listed.

In their functional model of design aid technology, Henderson & Cooprider (1990) argue that design aid environments mainly consist of three components: (1) production technology containing facility for representation, analysis and transformation of objects, relationships and processes; (2) coordination technology with control functionality supporting planning and enforcing rules, policies that will govern or restrict the design process and with cooperative functionality enabling users to exchange information relevant for the work process; and (3) organizational technology with support functionality to help users and with infrastructure functionality providing standards enabling portability of skills knowledge, procedures or methods across projects.

This article has presented an analysis of needs for computer support pertaining to the coordination technology component, and its relationships to the production technology component. CASE tools supporting the conceptualization of the software architecture, as well as test planning tools have been available for more than 10 years. These tools primarily provide functionality within the production technology component, i.e., support the individual test tasks, or provide an overview of the state of affairs by applying a specific set of testing metrics. Although certain coordination aspects are supported by these tools, several improvements are required. Furthermore, the coordination support must be integrated with existing tools and techniques, for example cooperative software inspection tools (Freedman and Weinberg, 1990; Mashayekhi et al., 1993). Although we have begun an identification of such support needs, much work still remains to be done before the ideas can be realized.

Future work should go in at least three directions. One is to evaluate related products and concepts, i.e., relate the support of the cooperative aspects to the individual aspects of software testing and reflect on how cooperative work can be computer supported. Henderson and Cooprider’s model will be relevant as a starting point for this work. The second direction is to tie our body of work to the ongoing efforts in conceptualizing relevant aspect of cooperative work, for
example the concept of Coordination Mechanisms (Schmidt and Simone, 1995), and the third direction will be to build experimental prototypes as a step forward in concretizing the ideas.

Acknowledgments

This research could not have been conducted without the invaluable help of numerous people at Foss Electric. Thanks to Leif Løvborg for many useful comments and suggestions, and to Susan Leigh Star for helping us understanding central aspects of classification schemes. We also thank Finn Kensing, Betty Hewitt, and anonymous IRIS’94 and Scandinavian Journal of Information Systems reviewers for constructive comments and ideas to previous versions of this paper. All errors in this paper naturally remain the responsibility of the authors.

References


Chapter 5

Object-Oriented Modeling of Coordination Mechanisms

Peter H. Carstensen, Birgitte Krogh, Carsten Sørensen

Systems Analysis Department, Risø National Laboratory, Denmark

Department of Computer Science, Aalborg University, Denmark

Computer support can enhance the effectiveness and efficiency of the coordination of distributed tasks. Analytical modeling of the work setting is an important prerequisite for designing computer-based coordination mechanisms. Object-oriented approaches to analysis, design and implementation have proven to be forceful means of modeling and analyzing complex systems. The aim of this paper is to experimentally investigate how object-oriented analysis can contribute to modeling coordination mechanisms. The experiment is based on the results from a field study investigating the coordination of distributed software testing activities. Previous identification and analysis of a set of linked coordination mechanisms is the starting point for the modeling experiment. These mechanisms consisted of paper and computer-based artifacts which were filled in and routed according to written procedures and culturally embodied conventions. The experiment resulted in an object model containing object-clusters, an object-class structure, and events and behavior diagrams. The experiment indicates that the modeling is useful for specifying the structural properties of coordination mechanisms as classes and objects. This facilitated decisions regarding how to link coordination mechanisms. The dynamic properties of coordination mechanisms reflecting interaction between actors are, however, not easily expressed in the models.

1. Introduction

Most work is complex and demanding, involving problem solving, decision making, rule interpretation, and cooperation. Decisions and activities must be conducted expeditiously and relate to new and constantly changing parameters, and it might require expanded capacities and special skills. This will often require involvement of several persons with different areas of competence. When people engage in cooperative activities, they become mutually interdependent in their work, and must, therefore, coordinate and integrate their individual activities to get the work done (Schmidt, 1991). The effort of coordinating complex and distributed work activities is often in itself complex and demanding. It is, therefore, obvious to consider how computer based coordination systems can

† Authors are listed in alphabetical order only.
support coordination activities—the core problem setting for research in Computer Supported Cooperative Work (CSCW) since 1984 (Bannon, 1993). In order to build computer based coordination systems, we need techniques and conceptual frameworks to support the identification and description of relevant structures and processes.

The theory of coordination mechanisms (in previous publications named “mechanisms of interaction”) provides a perspective and a set of concepts which can be applied in studies of cooperative work. A coordination mechanism is “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” (Schmidt and Simone, 1995). Furthermore, a computational coordination mechanism 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” (Schmidt, 1994).

The coordination mechanism concept can, amongst others, be viewed as a generic concept to describe artifacts stipulating subsets of who is doing what, when, where, how, and why. To illustrate the concept, let us consider a simple scheduling system supporting people in coordinating the scheduling of meetings by providing a standard calendar form which is routed among the participants, who enter suggestions for meetings and for meeting rooms. When meeting rooms or time-slots are double-booked by one of the participants, the relevant people involved will be notified in order to perform re-scheduling. We would claim that this system contains a coordination mechanism. The calendar form is a standardized artifact, perhaps with rows of dates and columns of available meeting rooms. It is symbolic in the sense that booking a room at a particular date only reserves the room, it does not change the state of the room itself. It contains a routing scheme, and perhaps also rules for how to fill in a form, i.e., it contains a protocol. This particular coordination mechanism reduces the complexity of distributed actors coordinating where and when to meet with whom—they would perhaps only need to engage in face-to-face meetings or telephone conversations in rare cases. The complexity of going from paper-based to computational coordination mechanisms is, amongst others, one of providing flexibility and local control with respect to the standardized formatted artifacts and the protocols.

Field studies have identified and analyzed a number of coordination mechanisms used in different real life work settings (Andersen, 1994; Pycock and Sharrock, 1994; Sørensen et al., 1994; Carstensen et al., 1995). The coordination mechanism concept has also been applied to analyses of a number of state-of-the-art CSCW technologies (Simone and Schmidt, 1993). Most recently, the first steps towards specifying a formal notation for coordination mechanisms have been made (Simone and Schmidt, 1994). In order to bridge the gap between field
Object-Oriented Modeling of Coordination Mechanisms

studies of cooperative work and design of coordination mechanisms, there is a need for suitable methods and techniques facilitating the modeling activities (Hughes, 1993).

The field studies indicated that coordination in real work settings often are supported by small isolated mechanisms each supporting a limited set of coordination activities, e.g., a form ensuring that all relevant information are distributed to the relevant actors, or a board providing an overview of state of affairs to the actors involved in a cooperative activity. Furthermore, the findings indicated that the different mechanisms are related to each other in different ways: A mechanism might trigger an action in another mechanism, a mechanism can ‘subscribe’ to information provided by another mechanism, a mechanism can ‘update’ the information in another mechanism, the procedure for the use of one mechanism can be defined by another mechanism, etc. (Schmidt et al., 1994; Schmidt et al., 1995). We can, thus, regard coordination mechanisms as individual detached and interrelated mechanisms providing a subset of the coordination means required to handle the coordination. The characteristics of on the one hand being detached mechanisms having “their own life”, and on the other hand being interrelated, could call for an object-oriented approach when modeling. The central techniques of abstraction, encapsulation, association relationships, etc. (e.g., Coad and Yourdon, 1991) in the object-oriented paradigm seems to be relevant to consider for modeling the aspects of coordination work to be computer supported.

This paper explores the question: What are the possibilities and limitations for modeling computational coordination mechanisms using an object-oriented analysis method? In order to explore this question, we have conducted an experiment applying an object-oriented analysis method in order to model coordination mechanisms identified in a field study. The analysis of the field study provided detailed descriptions of the coordination mechanisms used for coordinating distributed software testing activities. Based on this, we specified an object-oriented analysis model using the OOA&D method promoted by Mathiassen et al. (1993; 1994). The experiment resulted in an object model containing a system definition, a cluster structure, a class structure, an event list, and behavior diagrams. The OOA&D method was, among other reasons, chosen because it explicitly aims at grasping the dynamic aspect of the problem domain.

Others have addressed the problem of modeling the central aspects of cooperative work with the purpose of building computer support. Some are based on theories from other research areas and application of these (e.g., Flores et al., 1988; Malone and Crowston, 1990). Others have been mainly driven by technological possibilities (e.g., Kaplan et al., 1992; Swenson et al., 1994). Only few studies have, however, addressed the applicability of object-oriented modeling techniques for modeling the coordination aspects of a real life cooperative work situation.

The experiment indicates that the object-oriented modeling technique applied is useful for specifying the structural properties of coordination mechanisms as classes and objects. It can facilitate the process of designing coordination mecha-
nisms and their linking. The dynamic properties of coordination mechanisms reflecting interaction between actors are, however, not easily expressed.

First we discusses the research approach applied. Then the object-oriented approach is presented, followed by central characteristics of the coordination of distributed software testing, and by the experiment and the resulting models. The paper concludes with reflections on the usability and applicability of an object-oriented approach to modeling coordination.

### 2. Research Approach

The basic approach applied in this paper is an experiment in object-oriented modeling of coordination mechanisms, based on field study data. The field study analyzed the coordination of cooperative work in manufacturing settings where the participants dealt with the complexity of designing a complex instrument. We conducted, over a period of more than three months, more than 20 interviews, attended more than 10 project meetings, and spent 75 man-hours on observation. The effort covered both software engineering and manufacturing engineering (cf., Sørensen et al., 1994; Carstensen et al., 1995). This paper only uses the results from the software engineering part. The collection and analysis of data can be characterized as qualitative research inspired by Work Analysis (Schmidt and Carstensen, 1990) and ethnographic approaches (e.g., Bucciarelli, 1987).

There are, at least, two good arguments for choosing an object-oriented approach: Firstly, there are many indications that object-oriented methods will play a very important role in software development in the near future (Monarchi and Puhr, 1992). Thus, testing its usability in different situations is relevant. Secondly, the nature of the identified coordination mechanisms can, on the one hand, be regarded as individual detached mechanisms each supporting a limited set of coordination activities, and, on the other hand, the mechanisms often have to link to other mechanisms in order to get relevant information, trigger an action in another mechanism, etc. The principles of abstraction, encapsulation, and association structures (cf. e.g., Coad and Yourdon, 1991) seems to match some of the central characteristics of the coordination work identified in the field study.

We have chosen the OOA&D method by Mathiassen et al. (1993; 1994). Using the classification structure of Monarchi & Puhr (1992) the OOA&D approach is representative of object-oriented methods, providing both process- and representation support. Furthermore, some of the central characteristics of many coordination activities are the dynamics, and the facts that the activities often are related to several structures in the problem domain. OOA&D has explicitly addressed these issues.

The experiment was conducted over a period of four whole days where we developed the model by following the prescriptions in the method. This was then followed by several iterations of the model made through discussions with experts
in object-oriented modeling, etc. The outcome of this process is illustrated in section 5.

3. Object-Oriented Analysis and Modeling

A number of object-oriented methodologies for computer system design have been proposed. There is a large number of publications on the topic which has become increasingly popular in both industry and academic communities (Monarchi and Puhr, 1992). The methodologies (e.g., Coad and Yourdon, 1991; Jacobson et al., 1992; Taylor, 1992; Mathiassen et al., 1993) encompass principles and guidelines for the development of systems—which tasks to carry out, which techniques to use, etc.—as well as for what representations to produce in the process, i.e., which diagrams and descriptions to make. The object-oriented paradigm is intended to be applicable to all kinds of systems.

3.1 Object-Oriented Methods in General

According to Monarchi & Puhr (1992) there is little standardization in the field of object-orientation, apart from the notion of class and object. Some general characteristics can, however, be identified: The elemental concept of the paradigm is, not surprisingly, that of an object modeling a phenomenon of the real world. An important underlying principle of object orientation is that of being able to use the same set of concepts throughout these different phases or aspects of the development process. Other key principles are those of encapsulation, coherence, and reusability. In object-oriented methodologies, an object models an entity, or a phenomenon, in the real world. A class denotes a set of objects sharing static and dynamic properties. Structural relationships between classes are one of the main strengths of object-orientation. Relationships like one kind of vehicle being a specialization of another, or a customer having three bank accounts, are expressed in inheritance hierarchies, and association or aggregation relationships respectively. The expressive means and the resulting products are typically lists of objects and events, diagrams illustrating static and dynamic properties of single classes, and structure diagrams for interrelated objects. All with corresponding textual descriptions.

3.2 OOA&D

The modeling approach applied in this article is the analysis and design methodology, OOA&D, promoted by Mathiassen et al. (1993; 1994). It is based on selected techniques from a number of established object-oriented methodologies, together with basic principles for using the various concepts, techniques, and notation forms. The most important principles include: “using objects as a unifying concept” and “describing the model of the problem domain before the requirements to the functionality”. The former is what basically imbues all object-
oriented methodologies; having as key concepts: object, class, inheritance, association, aggregation. The latter is to be seen as a contrast to having functional requirements as the primary objective, i.e., basing the modeling of dynamic aspects on the concepts of event and behavior. This is inspired much by the (non-object-oriented) methodology, Jackson System Development (Jackson, 1983), since no existing object-oriented methodologies seemed to grasp the kind of dynamics of the problem domain, that was sought expressed when developing OOA&D.

Thus, specific focus, in the first part of the methodology, is on building a dynamic model of the problem domain. Among the main activities in the analysis phase, according to OOA&D, is the specification of a system definition stating what elements are of interest in the problem and application domains, including the general functionality of the system. Other central activities are the generation of a structure diagram, that expresses the static aspects of the model, and behavior diagrams, that define the relationship between events for each object class.

4. The Coordination of Distributed Software Testing

A field study was conducted at Foss Electric A/S, a Danish company manufacturing highly specialized equipment for analytical measuring quality parameters of agricultural products. We especially addressed the coordination of the software testing in one of their projects: The development of the S4000 instrument to be used for raw milk analysis. The S4000 consists of approximately 8000 mechanical and electronic components, and of around 200,000 lines of source code running on an integrated Intel-based 486 PC. The software handled the configuration and operation of the instrument through a Windows user interface. The software group involved in developing the S4000 consisted of between 5 and 12 members during the approximately 2 year development period. Other parts of the project included designers from the areas of mechanical design, electronic design, and chemistry, as well as people from the production, marketing, quality assurance, quality control and service departments. Many of these were also involved in testing the S4000 software.

In order to coordinate the testing activities, a standardized software-bug form and a centralized binder was used for registering and filing identified bugs. Stipulations of how to use the form was documented in a set of organizational procedures and in a set of culturally embodied conventions. The form and the related procedure are both illustrated in Figure 1.

The development and correction work was organized in phases called “platform periods”. A platform period was typically 3–6 weeks work followed by one week of integration. All the work and the plans were structured in relation to these periods. For each period a designer was appointed Platform Master
Several groups of actors (or roles) were involved in testing process: (1) Testers from different departments involved in the testing of the S4000 software; (2) the spec-team: A group of three software designers responsible for diagnosing the bugs and deciding how to handle the correction of the bugs; (3) a number of software designers each responsible for one or more software modules; (4) a software designer responsible for maintaining the central binder containing forms for all registered bugs; and (5) a software designer, called the Platform Master, responsible verifying the all corrections to the software during the platform periods, where each individual developer’s modules were integrated and compiled to an executable system. Furthermore, one of the spec-team members were responsible for continuously updating the work plans.

<table>
<thead>
<tr>
<th>Initials:</th>
<th>Instrument:</th>
<th>Report no:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date:</td>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td>Description:</td>
<td></td>
<td>(3)</td>
</tr>
<tr>
<td>Classification:</td>
<td></td>
<td>(4)</td>
</tr>
<tr>
<td>Involved modules:</td>
<td></td>
<td>(5)</td>
</tr>
<tr>
<td>Responsible designer:</td>
<td></td>
<td>Estimated time:</td>
</tr>
<tr>
<td>Date of change:</td>
<td>Time spend:</td>
<td>Test date:</td>
</tr>
<tr>
<td>[ ] Periodic error - presumed corrected</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accepted by:</th>
<th>Date:</th>
</tr>
</thead>
<tbody>
<tr>
<td>To be:</td>
<td></td>
</tr>
<tr>
<td>1) Rejected</td>
<td>2) Postponed</td>
</tr>
<tr>
<td>Software classification (1-5):</td>
<td>___</td>
</tr>
<tr>
<td>Platform:</td>
<td></td>
</tr>
<tr>
<td>Description of corrections:</td>
<td>(8)</td>
</tr>
<tr>
<td>Modified applications:</td>
<td></td>
</tr>
<tr>
<td>Modified files:</td>
<td></td>
</tr>
</tbody>
</table>

The actors fill (or add information) in:
- The testers: (1), (2), (3), and (4)
- The spec-team: (3), (4), (5), and (7)
- The designers: (6) and (8)

The procedure for handling bugs:
- A tester register and classifies a bug (field 1,2,3, and 4)
- The tester sends the form to the spec-team
- The spec-team diagnose and classify the bug (field 3, 4 and 7)
- The spec-team identifies the responsible designer (field 5)
- The spec-team estimates the correction time (field 5)
- The spec-team incorporates the correction work in the work plans
- The spec-team requests the designer to correct the problem
- The designer corrects the bug and fills in additional correction information (field 6 and 8)
- The designer sends the form to the central file
- The CFM sends the form to the PM and insert copy in central file
- The PM verifies the correction
- The PM returns the form to the central file

Figure 1: A translated version of the bug form and the procedure followed when using the forms. CFM is central file manager and PM is platform master. The form is a sheet of A4 paper printed on both sides. The figure illustrates who fill in the information in the form.

When a bug was identified, the tester filled in a form and sent it to the spec-team. The spec-team diagnosed the problem and decided which developer should fix the problem. The responsible designer were notified (by receiving a bug form),
and estimated the correction time needed. When the problem was dealt with, the designer notified the Platform Master who could then verify the corrections.

At any state, the binder contained a copy of the form in its current status. The binder had seven entries reflecting the status of a specific bug: (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). For each of the seven categories were the forms filed chronologically. The entries played a central role in stipulating the coordination by providing all involved designers and testers access to the state of affairs in the testing.

The work plans were organized in a large spread-sheet containing information on: which tasks are to be accomplished and a reference to a detailed description of the task, the estimated amount of labor-time per module for each task, the responsibility-relations between modules and software designers, the relationships between the tasks and platform periods, and the total planned work hours per platform period for each software designer. The work plans were maintained by the Platform Master and a member of the spec-team.

The means for coordinating the software testing activities were basically the bug form, the binder, the organizational procedures, and the 15 platform periods (for version 1 of the software). The actors, the tasks to be conducted, and the platform periods were meshed in the work plan spread-sheet. To support the integration of the software, the software architecture was reflected in a directory structure implemented on each designer’s work station. This structure can, thus, also be viewed as a means for coordination. To facilitate the actors’ awareness of the state of affairs, the binder and the spread-sheet were used. Since these means could not handle all the coordination needs, a lot of discussions, ad hoc meetings, and planned meetings, were conducted during the project, for example concerning classification of a bug, negotiation of an estimate or resource allocations, etc.

It seems sensible to think of the structures supporting coordination as detached inter-linked mechanisms supporting different aspects of the coordination. We will, hence, in the following attempt to model these coordination mechanisms by means of object-oriented techniques.

5. Object-Oriented Modeling of a Coordination Mechanism

The purpose of the experiment was to build an object-oriented model of the computational aspects of the coordination of software testing in the S4000 project. We have aimed at capturing the problem domain characteristics, and the modeling process, therefore, did not result in specifications of interface or functions. Since the fundamental principles of object-orientation includes that of using the same basic concepts throughout the system development process, the first step, the “analysis phase”, that determines the essential components in the system, i.e.,
Object-Oriented Modeling of Coordination Mechanisms

objects and classes, is where the applicability of object-orientation on a certain domain comes to a test. Therefore this contribution to the question of object-oriented modeling of coordination addresses purely the analysis aspects of system development.

The experiment consisted of the following activities, in consonance with the recommendations of OOA&D: (1) Specifying the System Definition; (2) identifying candidates for object-classes; (4) specifying clusters of classes; (4) identifying main attributes for each class; (5) specifying relationships between classes in a structure diagram; (6) identifying relevant events; and (7) specifying event-behavior for each class. The method does not prescribe that the model is specified according to a strict sequence, and the experiment consisted of several iterations each refining the model. This section presents the model resulting from the experiment.

5.1 System Definition

A system definition, according to the method, is a short and precise piece of natural language text that defines the scope and boundary of the system with respect to a number of crucial factors: The conditions under which the system is to be developed and used, if any such are of specific interest in that they, say, restrain the possibility for user involvement, or consist of areas of conflicting interests. The problem and application domain and the functionality, i.e., an overview of the phenomena (later modeled as objects, hence termed the object system) and tasks to be captured/supported, and how to support those by the system. These considerations constitutes the relevance criteria when assessing class, event, etc. candidates at more detailed levels later on in the process. The technology available during development and usage (this we have omitted, irrelevant considering the focus of the article). The philosophy, describing the essence and idea behind the system and how it is supposed to form part of the work setting. Below is the system definition for the modeling of the Bug Handling Mechanism:

The system is to be used by the testers, software designers, and management as an integrated part of the coordination involved in the cooperative work of registration, diagnosis, and correction of bugs. The object system consists of software and bugs, software testers and designers, the tasks they carry out with respect to bug correction, and the work plan for scheduling and coordinating these tasks. The corresponding application domain is the registration, planning, routing, and tracking of the whole bug handling process, and the functionality required is storage and retrieval of information on who is doing what and when. The systems philosophy is fourfold: automating the routing of tasks, making available state-of-the-bug information, reducing the complexity of coordinating the tasks, and strengthening actors’ self organization.

5.2 Classes and Structure

The set of relevant classes is subdivided into three clusters: Software, Work, and People. They encompass, respectively, the software that is being debugged and its
bugs, the debugging work tasks, and the people performing debugging tasks. The cluster view on the model is merely for overview purposes. What is more important are the generalization and aggregation relationships, expressing the range of possible static situations in the problem domain. Figure 2 shows the structure diagram, and in Table 1, each class and structure is explained in more detail.

Figure 2: The Structure Diagram. Triangles symbolize aggregation, and semicircles generalization structures. Lines with no symbols are associations. Numbers and intervals are the cardinalities, i.e., they state how many instances of the other class an instance of one class can be related to.
### The Software Cluster

**SW-architecture:**
The software part of the instrument, or the part of it, that is currently being debugged. It is an aggregation of a number of software modules.

**SW-module:**
Attributes(Developer)
A distinguishable part of the SW-architecture. At any given moment it is related to zero or some bugs that have been diagnosed to originate herefrom. Each SW-module has a Developer responsible for it.

**Bug:**
Attributes(Description, classification, testers initials, module relation, the various information resulting from diagnosis through verification tasks, current state)

*The* central class of the system. A bug is related to the tester that found it, and becomes associated with a classification and the suspected SW-module as part of the diagnosis process. At any given moment it is related to one, and only one, of the bug correction tasks and therethrough indirectly with an actor. It is the filling in and updating of a bug objects attributes and the dynamic change in its association with tasks that are the essential dynamic behavior of the system.

**Classification, Importance, and Risk:**
Basically these classes constitute the range of possible categorizations of bugs; each classification is a combination of the assessed risk and importance of a bug.

### The Actor Cluster

**Actor:**
Attributes(Name etc.)
An actor is any person who can take on one or more of the roles that are relevant for the bug correction process. Through the roles, an actor can be involved in zero or more tasks of the same or different kinds. For the sake of completeness of the work plan, an actor can also be related to any other tasks that are not relevant to the debugging, but that take up the persons work hours.

**Role:**
Attributes(Assigned actor and current tasks)
A role is taken on by an actor for some time, during which a number of tasks, of the specific kind that role is associated with, can be conducted.

### The Work Plan Cluster

**Work plan:**
Where information on scheduled work and resources is gathered.

**Period:**
A scheduling and coordination entity.

**Task:**
Attributes(Bug, actor, time)
Short for Bug Correction Task. Any task is always related to a bug and an actor. A task object models the actual state of a bug, as opposed to the intended state.

**Continuous Task:**
The kind of tasks that, for scheduling reasons, are *not* necessarily related to specific periods.

**Periodical Task:**
The ones that *are*.

**Diagnosis:**
The first treatment of an identified bug. During the existence of such an object, the bug is classified by a member of the SpecTeam and associated with the SW-module suspected of containing the bug.

**Estimation:**
Based on the diagnosis, a correction time is estimated by a developer.

**Correction:**
One or more developers perform the correction itself. Possible outcomes of this task, apart from the bug being corrected, is that it has to be diagnosed or estimated again.

**Verification:**
The bug correction is verified, i.e., it is either acknowledged as corrected and 'filed' or it has to be registered all over again.

**Other Task:**
Relevant only to planning the work because one of the actors is assigned to it.

---

Table 1: Descriptions of the classes, organized according to the three clusters. Details about attributes of no importance to the experiment are omitted.
Throughout the remainder of the article, capitalized words are used for the names of classes and clusters to help distinguishing them from other uses of the same words.

5.3 Dynamics

Events are the atomic parts of the dynamic aspects of the problem domain. That is, an event is any occurrence in the real world that causes a change in the value of the attributes of one or more classes in the object system, or that requires a change in the structure of the classes, typically an associations being created or removed.

<table>
<thead>
<tr>
<th>Event List</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>New Actor inserted</td>
<td>Bug rejected (no task association happens, bug deleted and/or re-registered later)</td>
</tr>
<tr>
<td>New SW-module inserted (and put into SW-architecture aggregation)</td>
<td>Bug accepted (a new Diagnosis is associated with the Bug and a SpecTeam Member)</td>
</tr>
<tr>
<td>New Period inserted (and put into Work plan aggregation)</td>
<td>Bug diagnosed (the Diagnosis is deleted, a new Estimation is associated with the Bug and a Developer)</td>
</tr>
<tr>
<td>New Importance inserted</td>
<td>Bug estimated (the Estimation is deleted, a new Correction is associated with the Bug and a Developer)</td>
</tr>
<tr>
<td>New Risk inserted</td>
<td>Bug corrected (the Correction is deleted, a new Verification is associated with the Bug and a Verification Responsible)</td>
</tr>
<tr>
<td>New Classification inserted</td>
<td>Bug re-diagnosed (the Correction is deleted, a new Diagnosis is associated with the Bug and a SpecTeam Member, when the corrector finds the diagnosis was wrong)</td>
</tr>
<tr>
<td>Actor assigned as SpecTeam Member (and put into SpecTeam aggregation)</td>
<td>Bug re-estimated (the Correction is deleted, a new Estimation is associated with the Bug and a Developer, when the corrector finds the estimation was wrong)</td>
</tr>
<tr>
<td>Actor assigned as Verification Responsible</td>
<td>Bug correction verified (the Verification is deleted)</td>
</tr>
<tr>
<td>Actor assigned as Platform Master</td>
<td>Bug re-registered (the Verification is deleted, a new Bug is registered, when the verifier finds something else was the matter)</td>
</tr>
<tr>
<td>Actor assigned as Tester</td>
<td>Bug deleted (when the information is obsolete)</td>
</tr>
<tr>
<td>Actor assigned as Developer</td>
<td></td>
</tr>
<tr>
<td>Actor resigned as SpecTeam Member</td>
<td></td>
</tr>
<tr>
<td>Actor resigned as Verification Responsible</td>
<td></td>
</tr>
<tr>
<td>Actor resigned as Platform Master</td>
<td></td>
</tr>
<tr>
<td>Actor resigned as Tester</td>
<td></td>
</tr>
<tr>
<td>Actor resigned as Developer</td>
<td></td>
</tr>
<tr>
<td>Actor assigned to Other Task</td>
<td></td>
</tr>
<tr>
<td>Platform Master assigned to Period</td>
<td></td>
</tr>
<tr>
<td>Developer made responsible for SW-module</td>
<td></td>
</tr>
<tr>
<td>Bug registered (a new Bug is instantiated and associated with the responsible Tester)</td>
<td></td>
</tr>
<tr>
<td>Bug postponed (no task association happens, bug deleted and/or re-registered later)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: The final list of events in the case. In parenthesis the occurrences that the event causes for each class are given in more detail in the cases where more classes are involved than are mentioned in the event name, or just for clarification. Details about which attributes are updated when are omitted.

For each class a behavior diagram is made, illustrating the constraints on the order in which the events that involve that specific class can occur. Behavior diagrams are diagrammatic representations of finite automaton. With the two representation forms, one can determine equivalent patterns of events that objects of a class has to follow. In the case of the bug handling mechanism, a large number of events are common for a number of classes, in that they involve changes in more than one type of object. Also, many classes are involved in more
than just a few events. These two characteristics tend to yield complicated behavior diagrams; the former in that it makes it difficult to get an overview of the consequences of one specific event, the latter in that many events has to be part of the same diagram. We have chosen to present one event and all the classes it involves as an illustrative example of how the dynamic aspects are modeled.

The sample event: Some of the most crucial events modeled in the system are those that have to do with the transition from one bug correction task to another. They are also the single events that involve objects from the largest number of different classes. We have chosen the event “Bug corrected” for the illustration of how the dynamics of the problem domain is modeled object-oriented. This event represents the situation where a correction task has finished successfully—i.e., without the need for re-diagnosis or re-estimation—and the relevant next task, the verification, is to take over. This event is relevant for the five classes: Bug (naturally), Correction (the task that just finished), Verification (the next task to take place), Developer (who did the correction), and Verification Responsible (who is to verify the correction). Four of those behavior diagrams are shown in Figures 3 through 5. The fifth diagram, for Verification Responsible, is omitted due to the fact that it is much like that for Developer, only simpler in that there are only three instead of six events in the selection.

![Behavior Diagram for Bug]

Figure 3: The behavior diagram for the class Bug. The events with a ‘o’ are part of the same selection. A ‘*’ is an iteration, and a row of events on the same level, that are either marked ‘*’ or not marked form a sequence.

“Bug corrected” is considered to be one, atomic, task because it is an important part of the coordination happening in the bug correction work setting, that one person finishing a task means the next person to take over can start – and is made aware of the fact that he can start – his own task on that specific bug. Six things happen when this sample event occurs, in terms of how the structure diagram and classes are changed: (1) Attributes in the Bug regarding the correction are updated; (2) the association between Bug and Correction task is removed; (3) the Correction task is deleted; (4) a Verification task is instantiated and (5) associated
with the Bug; and (6) the Verification task is associated with a Verification Responsible.

Figure 4: The behavior diagrams for the classes Correction and Verification.

Figure 5: The behavior diagram for the class Developer.

6. Discussion

The aim of this paper has been to explore the possibilities and limitations for modeling computational coordination mechanisms using an object-oriented analysis method. To do so, we have build an object-oriented model of a coordination mechanism supporting the coordination of distributed software testing.

In assessing the outcome of the experiment, it is, of course, vital to note that modeling, like many other human activities, involves a certain element of style. We are not in any position to claim that we are experts, but we did conduct the experiment from a solid base in terms of thorough field studies of the work system and a substantial modeling effort, based on other approaches than the object-oriented, prior to the experiment. The chosen method, promoted by Mathiassen et al. (1993) is only one of many object-oriented methods. The object model part of it is, however, not radically different from most other approaches. It conforms with the general results from evaluations of other object-oriented methods documented (e.g., Monarchi and Puhr, 1992; Hughes, 1993).

The OOA&D method provided good support for specifying the static aspects of the model, e.g., defining clusters, classes, instances, aggregations, and associations. But the dynamic aspects of the problem domain modeled seems to
“disappear” in the model. This is not a problem specific for OOA&D. Object-oriented methods, in general, do not sufficiently address “...the sequence timing and control of events and processes” (Monarchi and Puhr, 1992, p. 4). The experiment confirms Monarchi & Puhr’s observations. There are more object-oriented static models, than dynamic models. Monarchi & Puhr explain this by referring to the data-modeling origins of object-oriented methods. They point at several mechanisms for managing structural complexity in object-oriented models, but only at one mechanism for handling behavioral complexity—contracts and collaboration graphs, from Wirfs-Brock et al.’s (1990) method. As it is now, the dynamic aspects of object-oriented models are expressed in “attached” formalisms, such as state-transition diagrams, Petri-nets, etc. There is a need for a closer integration between static and dynamic aspects of object-oriented models. Apart from these general conclusions, a number of lessons have been learned:

Lesson 1: Object-oriented modeling improved the quality of the structural specifications. Prior to the OOA&D experiment, entity-relationship diagrams, state-transition diagrams and data-flow diagrams were the only means applied to express the properties of the coordination mechanisms. The cluster specifications, the Structure Diagram, and the cardinalities of class-connections improved our understanding of which structural aspects of the coordination mechanisms should be computer-supported.

Lesson 2: The level of detail in events must be in an appropriate form to reflect real life coordination. The ‘Bug estimated’ event is, for example, defined as: “The Estimation is deleted, a new Correction is associated with the Bug and a Developer”. Alternatively, each sub-event could be expressed as a separate event, but this would result in loosing the essence of the coordination which takes place when a developer estimates the time needed to fix a bug; the notification of a developer to take over and start the correction. It is the contiguousness that characterizes the coordination according to the organizational procedure, not the individual sub-tasks. The computational coordination mechanisms we are modeling should provide the facility of automatically conveying, to one actor, changes in the state of the mechanism induced by another actor (Schmidt, 1994), as reflected also in the system philosophy, stated in the System Definition (cf. Section 5.1).

Lesson 3: Classes must reflect conceptualizations used in the real life coordination. It could, for example, be tempting, instead of having four classes expressing different tasks, only to have a ‘Continuous Task’ and a ‘Periodical Task’ class, or maybe even only to have a ‘Task’ class. Then the diversity of the tasks could be expressed through attributes and the behavior of actors. However, given the composite events, mentioned above, this would result in loosing the essence of the coordination mechanism. In the behavior diagram, which event was the one relevant in a selection would depend on the value of an attribute, and getting an overview of the consequences of an event like ‘Bug estimated’ would involve looking at
two instances of the same diagram instead of two different diagrams. Both are undesired situations.

Lesson 4: Although clustering classes provided a straightforward way of subdividing the structure, the process did not help in designing linking between coordination mechanisms. Previous to the experiment, we defined four linked mechanisms. Two of these were identical to the ‘Work’ and ‘People’ clusters. The two others were the ‘Software’ cluster divided into a ‘Software’ and a ‘Bug’ mechanism (Schmidt et al., 1994). The three clusters in the model makes sense from an OOA&D perspective, but the four mechanisms express that the ‘Bug’ mechanism should be able to link to an existing repository containing conceptualizations of the software architecture and modules. Furthermore, the concern of modeling linking between coordination mechanisms resulted in classes which, from an OOA&D perspective, should be expressed as attributes to other classes. The contents of the ‘Classification’ class could, for example, be defined as an attribute to the ‘Software bug’ class. We do, however, view the ‘Classification’ class as providing a link to an external mechanism maintaining the classification structure.

Lesson 5: Interactions between actors involved in coordinating their activities seems to “disappear” in the models. When modeling coordination work, the interaction between actors is an important structural property of the problem domain. Protocols stipulating the coordination are essential (Schmidt and Simone, 1995). Although all the events related to the procedure of registration, diagnosis, correction, and verification are described in different diagrams, we cannot see how the documents are exchanged among the actors.

Lesson 6: Although reflected in the behavior diagrams, complex dynamics is difficult to model. Coordination mainly consist of activities meshing symbolic references to the problem domain, e.g., “have actor A fixed bug 7?”. This implies that events usually affects many classes. According to Mathiassen et. al. (1993), mutual events should imply reconsideration of classes and structures. This principle is, thus, potentially in conflict with the nature of coordination work. Further, the dynamics is difficult to grasp from the models, e.g., in order to assess the ‘Bug corrected’ event comparison of five different behavior diagrams is required.

The conclusions and lessons discussed in this paper should, of course, be seen as only one measure towards establishing a useful set of modeling techniques. Further design and coding of a coordination mechanism for software testing can provide more detailed information of the usefulness of object-oriented methodologies for modeling coordination mechanisms. Also, the applicability of contracts suggested by Wirfs-Brock et al. (1990) needs to be addressed in further research.
Acknowledgments

This research could not have been conducted without the invaluable help from numerous people at Foss Electric, especially Ole Pfug, Jørn Ørskov, and Carsten Palludan. Thanks to Henrik Borstrøm who participated in the field study. We also thank Peter Axel Nielsen for valuable comments and suggestions. All errors in this paper naturally remain the responsibility of the authors.

References


Mathiassen, Lars, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage (1994): Objektorienteret design [Object-oriented Design]. Department of Computer Science, Aalborg University, Fredrik Bajers Vej 7, DK-9220 Aalborg Ø.


Abstract: This paper presents a prototype of a system supporting software designers and testers in coordinating the process of reporting, diagnosing, and correcting software bugs. The system is called BRaHS (Bug Reporting and Handling System). The system can be regarded as a computer-based coordination mechanism that specifies the flow of the work and mediates relevant coordination information among the involved actors. Taking departure in empirically based requirements the functionality and structure of the system is described. The system is discussed and characterized by means of the central concepts in the conceptual framework of Coordination Mechanisms: The three layered structure, the concepts of active artifacts, objects of articulation work, the linking between mechanisms, and the overall requirements of local control, malleability, visibility, and flexibility, etc.

1. Introduction

A series of COMIC deliverables, papers, reports, etc. have introduced and discussed the conceptual framework of Coordination Mechanisms (cf. e.g., COMIC deliverable 3.1, 3.2, and 3.3). These have described the framework (e.g., Schmidt et al., 1993; Schmidt, 1994c; Schmidt, 1994b; Schmidt and Simone, 1995), used the framework for analyzing artifacts supporting coordination work (e.g., Carstensen, 1994; Carstensen et al., 1995a), and discussed a notation for constructing computational coordination mechanisms (e.g., Simone et al., 1994a; Simone et al., 1994b).

This paper describes a next step in the process of developing the ideas and concepts in the framework. A prototype of a computer-based coordination mechanism, called BRaHS (Bug Reporting and Handling System) is presented. The aim is to establish a basis for a discussion of how the framework of Coordination Mechanisms can be applied in actual systems design, and produce input for improving and refining the framework.
BRaHS has been designed on the basis of a set of requirements derived from a field study (Carstensen, 1994; Carstensen and Sørensen, 1994a; Carstensen and Sørensen, 1994b; Carstensen et al., 1995b). The requirements have been discussed and analyzed in terms of functionality, expected use of the system, etc. The requirements have then been the driving force for the design, rather than following the ideas in the Coordination Mechanism framework. After designing and implementing BRaHS have we analyzed and discussed the prototype in terms of the concept of Coordination Mechanisms. In this paper the prototype is discussed in terms of functionality at the α, β, and γ-level, in terms of active artifacts, in terms of which objects of articulation work can be found in the prototype, etc. We further discuss how the overall requirements for coordination mechanisms concerning malleability, local control, visibility, linking to other mechanisms, etc. are reflected in the prototype.

In a design, like the one presented here, a lot of decisions have to be taken. Many of these is, of course, debatable one way or another. We have in this paper chosen only to discuss a few of these: When it is obvious to point at “better”, but much more demanding, solutions. The main purpose of the design has been to illustrate, rather than to refine, debate, discuss, iterate, etc. until a “perfect” design has been establish.

In order to provide a basis for understanding the rationale behind the design we first briefly describe the work setting in which the field study was conducted, and which overall requirements for computer support were derived. We then describe the prototype from users perspective, i.e., we describe the functionality in terms of what it offers, and illustrate the intended use of the system. In section 5 we briefly introduce the database structure of the prototype. Section 6 and 7 discuss the prototype in terms of a coordination mechanism, first the structure, and then the overall requirements for coordination mechanisms are examined.

### 2. The work setting and the needs for computer support

We have conducted a field study of a group of software designers and testers involved in a large development project at Foss Electric. Results from this study have been used for establishing requirements for computer support.

Foss Electric develop, produce, and market equipment for analytical measuring quality parameters of agricultural products. Equipment for measuring quality parameters of agricultural products is a highly specialized field. There are only a few companies in the marketplace, and Foss Electric is the largest in the world. The Foss Electric employs approximately 700 people. The products manufactured are used for measuring the compositional quality of milk (the fat content, the count of protein, lactose, somatic cells, bacteria, etc.), the composition and microbiological quality of food products, and for measuring grain quality. The measurement technologies are typically infrared, fluorescence microscopy, or bac-
teriological testing. Innovation towards new, better and faster measuring techniques is the most important strategic goals for the company. Research and development are thus essential activities. The organization is very much structured in terms of projects. These projects typically includes specialists with competence in fine mechanics, hardware and software design. In some projects also specialists in optics and chemistry are involved. Our field study concentrated on one of the large projects Foss Electric has recently accomplished, the System 4000 (S4000) project.

2.1 The S4000 project

The objective of the S4000 project was to build a new instrument for analytical testing of raw milk. Compared to previous instruments for testing milk, the S4000 system introduced measurement of new parameters in the milk (e.g. urea and critic acid), and the measurement speed was to be improved compared to previous products. The S4000 was the first product with an Intel-based 486 PC build-in. The configuration, control, and operation of the instrument should be made via a Windows user interface, i.e. a graphical user interface and use of mouse and keyboard. 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. More than 50 people have been involved in the project, which lasted approximately 2 1/2 year (for version 1 of the S4000 system). The core personnel involved in design included a number of designers from each of the areas of hardware design, electronical design, software design, and chemistry.

2.2 The software design group

During the project a group of between five and twelve software designers were working on designing, implementing, and testing the software. To handle the complexity of coordinating the process of testing, bug reporting, bug diagnosing, and correcting the software they organized the work in relation to different roles:

(1) Software designers.
All software designers were working as designers, i.e., they were responsible for designing, implementing, maintaining, and correcting bugs in one or more of the software modules. Hence, when a development task or correction task was related to a specific module the designer became ‘automatically’ responsible for it.

(2) The spec-team.
The spec-team was a group of three software designers responsible for diagnosing reported bugs and deciding how to handle each bug. The members of the spec-team in the S4000 project were appointed so that all the three main “layers” (the user interface, algorithms used for computing
the measuring results, and software interfacing to the network) were represented.

(3) Platform master.

The software design was organized in working periods called “platform periods” (cf. Carstensen, 1994). At the end of each platform period was the platform master was responsible for managing and coordinating all the activities involved in integrating the outcome of the working period. He was, among other things, responsible for verifying the corrections to the software made by the designers, i.e., control that the reported bugs had been conquered. The platform master was always one of the designers in the project, and the role were taken alternately by the software designers involved in the project.

(4) Project plan manager.

In most of the project period the software group did not have a dedicated manager. Instead they appointed one of themselves as responsible for maintaining a project plan spreadsheet. This could be regarded as a rolling project plan for the software development. The sheet contained information on: (1) Which tasks there were to be accomplished and a reference to a detailed description of the task; (2) the estimated amount of time per module for each task; (3) the responsibility relations between modules and software designers; (4) the relation between the tasks and which working period (platform period) they were planed to be finalized in; (5) and the total planned work hours per platform period for each software designer.

(5) The testers.

Testers were the actors involved in the concrete testing of the software embedded in the S4000 instrument. The testers could be affiliated in several different departments at Foss Electric. They had thus a very different background and approach to what functionality the software should provide, and what the most important characteristics of the software were (e.g., usability, stability, correctness, maintainability, etc.). They were typically software, hardware, or mechanical designers involved in the project, or they were employed in the departments of quality assurance, marketing, service, maintenance, etc.

(6) The central bugs file manager.

At any given point in time of the S4000 project, one of the software designers was responsible for organizing and maintaining the central bug file, a ring binder containing copies of all reported bugs and organized according to their status. Ahead of each integration period the central file manager was responsible for informing the platform master on which bugs had been reported as corrected since the previous platform integration period.
2.3 Coordinating the bug handling

As mentioned one of the important tasks of the software designers were to coordinate the process of registering, diagnosing, and correcting bugs identified during the tests. They invented and used a standardized form that all testers had to fill in whenever they identified an error (a bug). To prescribe the use of the forms a structured ring binder (being used as a central file) and a set of procedures and conventions (protocol steps) for the use of the form was established. Some of these were written down as organizational procedures. Others were conventions developed during project.

The purpose of the form and the protocol was to 1) support a decentralized registration of bugs, 2) support a centralized decision making on how to overcome the identified problem, 3) support the correction activities in being handled in a decentralized manner, 4) provide an overview of the state of affairs (with respect to registered, corrected, verified, etc. bugs), both to the involved software designers and testers, and to the management of the software development, and 5) support a centralized process for verifying the implemented corrections.

The form and a brief description of the procedural steps that was used when using the bug form are illustrated in Figure 2-1. For a further description of how the bug form is used see Carstensen (1994).

<table>
<thead>
<tr>
<th>Initials:</th>
<th>Date:</th>
<th>Instrument:</th>
<th>Report no:</th>
<th>(1)</th>
<th>(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classification:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1) Catastrophic 2) Essential 3) Cosmetic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Involved modules:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Responsible designer:</td>
<td>Estimated time:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date of change:</td>
<td>Time spend:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Periodic error - presumed corrected</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accepted by:</td>
<td>Date:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To be:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1) Rejected 2) Postponed 3) Accepted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software classification (1-5): ___</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description of corrections:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modified applications:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modified files:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The actors fill (or add information) in:
- The testers: (1), (2), (3), and (4)
- The Spec-team: (3), (4), (5), and (7)
- The designers: (6) and (8)

The procedure for handling bugs:
- A tester register and classifies a bug (field 1, 2, 3, and 4)
- The tester sends the form to the Spec-team
- The Spec-team diagnose and classify the bug (field 3, 4 and 7)
- The Spec-team identifies the responsible designer (field 5)
- The Spec-team estimates the correction time (field 5)
- The Spec-team incorporates the correction work in the work plans
- The Spec-team requests the designer to correct the problem
- The designer corrects the bug and fills in additional correction information (field 6 and 8)
- The designer sends the form to the central file
- The CFM sends the form to the PM and insert copy in central file
- The PM verifies the correction
- The PM returns the form to the central file

Figure 2-1: The bug form used at Foss Electric and the general procedure followed when using the forms for registering, diagnosing, and correcting bugs. CFM means ‘central file manager’ and PM is ‘platform master’. The form is a sheet of A4 paper printed on both sides. The numbers indicate who fills in which information in the form. The figure is similar to one shown in chapter 4 of this deliverable.
2.4 Requirements for computer support

Based on the analysis of the bug form and its use have we earlier established a set of requirements for a computer-based coordination mechanism supporting the coordination of the process of registering, diagnosing, and correcting software bugs (Carstensen and Sørensen, 1994b; Carstensen et al., 1994; Carstensen et al., 1995b). To summarize these, a computer-based coordination mechanism must provide:

- access to descriptions of every individual registered bug. The descriptions must contain information on originator, symptoms, priority, suggested diagnosis, involved modules. After the diagnosis has been made information on estimated correction time, responsible designer, status for correction is needed. Also references to the code and documentation is needed too.

- access to, and overview of, existing plans. Information on the use of both human and technical resources must be available. The relationships between tasks, deadlines, actors, software modules, etc. must be visible and modifiable.

- modifiable data structures that
  - reflect the architecture of the software complex being tested. The structure must reflect the main functionality of each module, relevant classifications of the modules (e.g., importance), and relations between the modules.
  - contain logical references from the architecture to the responsible software designers, relevant documentation, and the actual code.
  - contain aggregated information on registered bugs with respect to modules, types of bug, priorities, and responsible designers.
  - contain information on all involved actors, technical, and hardware resources involved in the development and testing. Both individual characteristics, and present and planned workload of the actors and technical resources must be available.
  - contain information and descriptions of the present classification categories for bugs, software modules, etc. The present classification categories must, of course, be reflected in all situations where the categories are used (e.g., in bug registrations).
  - include information on organizational procedures, techniques, standards, etc. used in the test work.
  - contain information on the requirements for the software complex, both as they are stated in existing specifications, and the interpretations suggested at the moment.

- support for distributed registration and classification of bugs, and support for the testers in filling in all required information for the diagnosing process.
- access to specify a work flow process stipulating to whom information on the bug must be send, how the bug is made visible to other testers, etc. The protocol stipulating the flow must be modifiable.
- functionality that automatically make the involved actors aware of the revised plans, obligations, etc.
- a channel for structured communication (e.g., structured messages) regarding distribution and negotiation of information. This must be used for negotiating diagnoses, resource allocations, deadlines, etc. The structure of the messages and the dialogue must be tailored to the concrete situation.
- a feature supporting the involved actors in suggesting changes to the categories and distribute these to other relevant actors, who can then reply to the suggestions.
- features supporting the testers and designers in making others aware of registered bugs and implemented corrections. Both user-activated and automatic distribution of this type of information must be available.
- access for all involved actors to, upon request, receive information on the state of affairs, both with respect to registered bugs, their diagnosis, and status, and with respect to the test and correction tasks.

This list should not be regarded as exhaustive. It is rather a first an very rough list of the most important requirements derived from the analysis of the field study data. For a more thorough and coherent listing of requirements see COMIC deliverable 3.2 (Carstensen and Sørensen, 1994b). The requirements will not be discussed further here.

3. The overall functionality provided by BRaHS

In order to illustrate ideas for how a computer-based system could fulfill the requirements listed above have we designed an illustration prototype. The effort has mainly been put into illustrating how the registration of bugs (improved classification of bugs) could be handled, how the overall requirements of malleability and local control could be reflected in the user interface, and how the protocol can be made visible and accessible to the users. Furthermore have we focused been on how the basic data structures in the database must be organized in order to support the needs for flexibility and making changes to the protocol, the classifications used, the roles involved, etc.

The prototype is implemented as a client/server structure using Borland’s Delphi as the application development and runtime environment. It is running on a Windows platform.

In this section will we illustrate the functionality and intended use of the system, i.e., illustrate what the system offers to the users and how it is offered.

The main purpose of BRaHS is to support a distributed registration of bugs, and to support an automatic routing (forwarding and passing on) of the
information to the next actor (role) in the tester -> spec-team -> designer -> platform master chain. Since an overall requirement for coordination mechanisms is that the control of the work flow must be in the hand of the user BRaHS should also access to changing the flow. BRaHS furthermore aims at providing the users with facilities for getting an overview of which bugs are reported, what are their status, etc.

To fulfill these purposes and requirements a series of overall functionality have been designed and implemented. These are:

- Log-in procedures where the user must identify him- or herself to the system and specify the role(s) he or she has when interacting with the system.
- Three windows for registering information about bugs. Both the testers, the spec-team, and the designers fill in information about a bug.
- A window for specifying the classification of a bug.
- Procedures that, when a user indicates that the registration is finished, automatically routes information on the bug to the next actor (role) in the flow.
- A graphical representation of the protocol specifying the information flow, and access for the user to make changes to the protocol.
- Facilities to search for registered bugs with certain characteristics, and facilities for browsing in the registered information.
- Procedures to set-up new projects, i.e., to establish a new (empty) database for registering bugs in relation to a new project.
- Procedures to define or change which actors, modules, roles, instruments must be involved in a project.

Each of these will be described in some more detail in the following.

3.1 Log-in

The log-in window is illustrated in figure 3-1 below. The main purpose is to provide some access control, and for the system to have an identification of who the user is and which role(s) he wishes to do.
This log-in procedure is very simple and primitive. When the system is going to be used in a real work situation this procedure must be redesigned carefully. First of all should a relevant level of security be reflected. Also the fact that the user here must indicate his role(s) beforehand should be reconsidered. In actual work situation it is very often the case that an actor switches between several roles. A designer might for example be tester in one situation and designer a few seconds later. Thus switching between roles must be reconsidered. This has only been supported through the possibility of indicating more than one role at log-in, and by providing a command button for changing role in the icon button row. This will probably be very insufficient and interrupting for actors in real work situations. It has however not been considered essential in this prototype.

After having logged-in the user gets access to a number of menus and functional icon buttons as illustrated in figure 3-2. The menus will be described later. The icon buttons give access to establish define which project should be opened. To create a new project (and thus a new database containing bug reports) is considered a facility that not all users must have access to. This is thus to be done via another application offering facilities for initiating new actors, roles, protocols, classification structures, etc. (cf. section 3.3 and 3.5)

When a project (and thus a database containing registered bugs) is opened the next four icon buttons give access to functions that: creates a new empty bug form, search for a specific bug form by means of the number, search for reported bugs fulfilling certain characteristics, and establish a list of all registered bugs from which the relevant can be chosen. The right most button is used for changing role.
3.2 Registration of information on bugs

Registering data about specific bugs is essential, both in the actual testing and correction work and in the prototype.

Several different actors (roles) are registering data about the bugs: Testers register information on how the problem have occurred and its importance. The spec-team register information on importance, deadlines, who is responsible for correcting it, and an estimation of correction time needed. The designer adds information about which corrections have been made, which modules and files are affected, and how much time have been spend on correction.

To provide this functionality have we designed a registration window organized as a three layered “index card” as illustrated in figure 3-3 below. The content of the registration window is the same for the tester, the spec-team and the designer, but in each situation fields not relevant will be ghosted.

On top is the index card to shift to a window used for classifying bugs, or to a window in which the user can get an overview of the protocol flow. These two facilities will be described in the following two sections.

The buttons in the bottom of the window is used for navigation, etc.
BRaHS: A Computer Based Mechanism Supporting the Coordination of Bug-handling

Figure 3.3: The first page of the three layered index card used for registering data about bugs. The report is separated into three chunks: one to be filled in by the tester reporting a bug, one to be filled in by the spec-team diagnosing the problem, and one to be filled in by the designer when the problem has been dealt with.

The entry fields are grouped into three. The upper group is used by the testers to register date, who they are, and which instrument they used when testing.

In the middle group must the spec-team enter (or select from selection lists) whether or not the problem reported can be accepted as a bug, in which platform period (working period, cf. Carstensen, 1994) it must be corrected, which module is expected to cause the problem, who is responsible for correcting the problem, and how much time is it estimated to take. Also a general free-text comment can be added.

The bottom group of fields are filled in by the designer after he or she has corrected the problem. He specifies when the bug has been corrected and tested and the amount of time spent on the correction. Furthermore he enters a free-text description of the corrections made and which modules and files it has affected. This field will also contain an identification of who has conducted the actual correction work. When the system is to be developed for real-life use the structure of the description field must be considered further. It shall probably be divided into several fields (selection-lists) in which the designer can indicate affected modules, affected files, types of correction, expected implications for other modules, etc.

As mentioned the buttons at the bottom of the window is used for navigating and controlling the interaction. The lock is used to lock the current record so that other users cannot make parallel updates. The forward and backward keys are
used to browse in the selected (opened) bug reports. The check mark button is used for saving the entered data, without finishing the registration. The Send button is used to indicate that the user has finished his registrations. When this is pressed the window is closed and the actor next in the work flow is notified. Cancel and Help should be self explanatory.

3.3 Bug classification

To support the coordination activities involved in managing the bug registration and correction process it is very important that both the tester and the spec-team make a proper and detailed classification of the problem. The classification scheme provided in the original paper based form was very primitive (cf. figure 2-1). In BRaHS have we chosen to classify bugs according to two dimensions: One having to do with which aspects of the product it concerns, e.g., the usability or the maintenance of the product. The other regards the importance of getting the problem corrected with respect to each dimension of concern. The window offering this is shown in figure 3-4 below.

An overall requirement for coordination mechanisms is that they must be malleable. Due to this is it implemented so that both the list of concerns and the importance scale can be re-specified by the users them-selves. This means that in a project were other dimensions are required (because it is a completely different product) can the users just specify which dimensions that should be used here. These corrections or expansions can also be made in an ongoing project.

Corrections to the classification scheme used in an ongoing projects have been considered both rare and important to have good control off. We have thus chosen that maintenance of the classification dimensions can be done through another application (the setup-application) only. In terms of the Coordination Mechanisms framework we consider this a β-level application.
Figure 3-4: The window used for classifying a bug. Both the tester, the spec-team, and the involved designers can enter as many ‘x’s in the matrix as they want to. For each ‘x’ will it be identified who has entered the ‘x’ (OJ, who is a designer in the example) and an optional additional comment. Furthermore will the free-text description field on the right side of the window contain a general description characterizing and describing the problem in further detail. This will typically be filled in by the tester, but also spec-team members, designers, and the platform master can add information.

All actors involved in reporting, diagnosing and treating the problem can at any time include a more detailed free-text description of the problem.

The navigation buttons are the same as those described in the previous section.

### 3.4 Routing facilities

Another overall requirements was that the system should provide automatic routing facilities. This is designed so that, when the one user press Send in order to finish his registration, the system up-dates the database with the entered data, and checks the protocol related to the bug form (described in the following section). From the protocol can it be seen which role must receive the form (the information on the bug) next. It is checked if this role is logged-in to the system. If so he will receive a message saying that bug no. x has been send for him. The receiver can the immediately open it a window and see the information registered so far, or he can decide to look at it later. If the receiver is not logged-in the system will send him a notification when the he log-in to the system.
3.5 Changing the protocol

Visibility of the protocol embedded in the mechanism, and possibilities for the local users to deviate from the protocol are two very essential requirements for coordination mechanisms. To deal with these in BRaHS have we included a index card layer in the registration window which graphically presents the protocol and in which the use can make changes to the protocol. The window is shown in figure 3-5.

![Figure 3-5: The window used for getting an overview the protocol flow, and for making changes to the flow. Each bug form can has its own flow, i.e., there is a protocol for each. Each icon in the diagram represents a role: The question-mark is a tester, the tool-set is the spec-team, the gear-wheel is a software designer, and the document is the platform master. The non ghosted icons illustrates the current protocol, and the arrows between them indicates the direction of the flow. The current status is marked by a red ring around the actual icon (cannot be seen on the black/white picture). The ‘Change flow’ provides access to manipulate the protocol. ‘Broadcast change’ sends notifications to other actors involved in the treatment of the actual bug that the protocol has been changed. ‘Hide/Show deviations’ remove/add the rings indicating the default protocol, and ‘History’ gives the user access to see how the protocol has previously appeared.

The idea of window is to provide the users with access to an overview picture of the protocol. Each icon in the diagram represents a role: The question-mark is a tester, the tool-set is the spec-team, the gearwheel is a software designer, and the document is the platform master. By clicking an icon can the user get further information on who actually has this role for the current bug. If for example the user presses the gearwheel icon information on the responsible designer will pop-up next to the icon. The non ghosted icons illustrates the current protocol having a flow as indicated in by the arrows. The ghosted icons and the broken arrows
shows the accepted deviations to the protocol the user can specify. This is probably a controversial implementation of local control, since the users are only allowed to make certain pre-specified deviations. An alternative solution would be to let the user have access to draw new arrows from whatever icon (role) to whatever icon he wants. In BRaHS have we chosen to limit the freedom of the users for simplicity reasons. This must, of course, be considered further in a real-life design situation.

The flow is changed by first pressing the Change flow button. Then the protocol is changed by clicking on the components. If the user clicks on a ghosted icon or broken arrow the icon or arrow is made active (non ghosted) and vice versa. When the user has finished making corrections he press the update (check mark) button. The system then checks the consistency of the protocol before saving it in the database.

When a new version of BRaHS is instantiated—if for example a new project is established—a default protocol is defined at the same time the actors and roles are defined. Whenever a new bug is registered it will be routed according to this default protocol until a user changes it. The default protocol is highlighted is Show deviations is activated. This information can be removed from the diagram by pressing the Hide deviations button. The user have thus access to see both the actual protocol, possible deviations, and the default flow serving as the standard protocol.

The user making changes to a protocol can chose to inform the other actors involved. This is done by pressing the Broadcast change button. When this is done all actors involved in the treatment of the current bug form will receive a notification saying that the protocol for bug no. x has been changed, and they can on demand see the changed protocol. This facility is only partly implemented in BRaHS.

The last facility to be mentioned is the history facility. On demand the user can browse in previous versions of the protocol related to the treatment of a specific bug. If the History button is pressed a series of new windows pops up. Each of these contains a diagram of an older version of the protocol for the current bug form. These diagrams can, of course, not be changed.

### 3.6 Searching and browsing

To fulfill the requirements users of BRaHS must also have access to facilities for finding information on specific bugs, or finding bugs fulfilling certain specific characteristics. Three different search facilities have been implemented.

The simple search allows the user to find a specific bug form by entering the bug number. When this is done the system will show information on the bug in a three layered index card window similar to those shown in figure 3-3, 3-4, and 3-5.
Figure 3-6: Three different search facilities are implemented. These can be accessed either via the search menu or via the three right most icon buttons on the opening screen.

Another quite simple search facility is the 'Show all'. This results in a list of all the bugs registered on the current project. For each bug is listed all information registered in the database, i.e., the bug number, an indication of whether the protocol has been changed or not, the acceptance state, the test date, the state of the bug, diagnose date, the status of the treatment, an identification of the tester, the designer involved, etc.

Figure 3-7: A show all search results in a list of all bugs registered on the current project. The user can by clicking on a specific line get detailed information on the individual bugs. The contents of each line is described in the body text of this section. The upper right corner of the screen contains information on the project (here S4000) and on the actual user and his role(s). In the lower left corner is included information on the total number of bugs registered on the current project.

The most complex search is what we in the menu have called advanced. If this is selected the user gets a window similar to the three layered index card window used for registering information on the bugs. Here can the user then specify the a search profile. The profile can thus be all kinds of combinations of all the fields
registered for all bugs. That is, the user can search for bug reports to be corrected in a specific platform period, bug reports for which a specific designer is responsible, bugs which it has taken more than 15 hours to correct, bugs having a specific classification profile, bugs that are treated according to a certain deviation from the standard protocol, etc., etc.

Figure 3-8: In the complex search all kinds of search profiles for all fields registered on the bugs can be combined. All fields can contain a combination of several selections. This is the main reason why all selection lists are shown in an additional window. This also minimize the requirements for how much the users must remember on his own. The concrete example will result in a list similar to the one shown in figure 3-7 containing all registered bugs to be corrected in platform period A5 in the CV Driver module by designer JY.

3.7 Set-up of projects, actors, modules and roles

As mentioned earlier a new project having its own actors, classifications, module structure, and default protocol must be specified in another application which the users usually will not have access to.

Some more local corrections to the way BRaHS is executed can however be made. New actors, modules, and instruments can be added to the lists, or the existing actors, modules, and instruments can be re-specified or removed from the lists.
The situation is very often that several actors have the same role, that one actor can have several roles, and that it will be obvious to group the modules and instrument components. Facilities for this is provided via a pull-down menu too.

4. A scenario for use of BRaHS

The previous section have given a rather detailed overview of the functionality provided by the system. To relate this to a use situation this section will briefly describe a small simple scenario of a possible use of BRaHS.

Mr. T is involved in the test of the software controlling the pipette of System 4000. He then discover that if the pipette meets two empty milk test glasses on the conveyer it stops. He checks it in the requirement specifications and realize that this is not right according to the specifications. This must be reported as a bug. Mr. T log-in to BRaHS and identify himself as tester. He opens the S4000 project via the Bug menu. He then creates a new bug form by pressing the New bug icon. An empty bug registration window is then opened (cf. figure 3-3).

Mr. T now fills in information on the problem. He classifies it according to support as a 3 (not too important) and according to usability and salability as an 8
(important). In the comments field on classification he writes a short notice saying that the conveyer will often contain empty glasses, so two glasses in a row will be quite common in actual work situations (cf. figure 3-4). He then press the Send button and the window is closed down.

Two days later has the S4000 spec-team their weekly meeting. Mr. S log-in to BRaHS as a spec-team member. He then search for bug forms having the status registered. The first on this list is the bug form filled in by Mr. T. He opens it and the spec-team members read the description. They quickly agree that this is a problem that must be related to the pip-control module which Mr. D is responsible for. Thus Mr. S tip of the accepted button, and select the pip-control module and Mr. D from the selection lists. The spec-team further more agree that it is important to have this correct in the current platform period so Mr. S enters an A4 in the platform field. They have some problems in estimating the correction time so the field is left open. The spec-team don’t add anything to the classification. The default protocol prescribe that the form on a diagnosed bug must be send to the tester before it is routed to the designer, cf. figure 4-1. But in this situation is the problem quite clear so Mr. S change the protocol so that the form is directed directly to the responsible designer, Mr. D.

![Figure 4-1: The default protocol prescribe that the form on a diagnosed bug must be send to the tester before it is routed to the designer: the arrow goes ‘down’ from the spec-team (the tools icon) to the tester (the question mark). This is not needed. Mr. S can change the protocol, so that the flow follows the icons in the upper line, i.e., it goes directly from the spec-team to the designer (the gearwheel icon) and further to the platform master (the hand and paper icon).](image)

When Mr. S presses Send the database is updated and a notification is send to Mr. D. Mr. D is a curious person so he opens the bug form right away and look at the description. He realizes that the problem is correct and that he must correct
when he has time to do it. He estimates it to take 12 hours to correct. This is entered and he updates the database and close the window.

A week later does Mr. D have time to look at problem and correct it. When he has finished doing this he logs in to the system, opens the actual bug form, enters the date for changing and testing, and fills in that he has spend 23 hours correcting it. Furthermore he writes a note saying that only the code in the pip-control module has been affected, and that he has changed the stop conditions so that an indefinite number of empty glasses are accepted without stopping the conveyer.

Mr. D has corrected the problem an all informations are now filled in.

Mr. D now press the send button to indicate that he has dealt with it. The database is update and a notification is send to the platform master, Mr. P.

Two weeks later, the day before the next integration period, Mr. P log-in to BRaHS and search for all bugs having the status of corrected but not yet verified. As a preparation to the forthcoming integration period prints Mr. P the description of each of bug on the list. He is now ready to check if all the problems have been dealt with properly.

5. The Database structure of BRaHS

The previous sections (2-4) have mainly been addressing the design of BRaHS from a user perspective, i.e., focus has been on requirements for which functionality and data structures a computer based mechanism must provide, and how these have been made accessible. The rest of this paper will shift towards analyzing BRaHS as a coordination mechanism.
So, although a detailed description of the architecture of the system is considered out of scope of this paper, it is relevant to illustrate how the database structure in BRaHS are designed. The reason for this is that several of the decisions taken in relation to the conceptual database design are closely related to the requirements of visibility, malleability, and local control. Other decisions have close relations to the architecture related to the concept of coordination mechanisms (cf. COMIC Deliverable 3.3: Simone et al., 1994a).

An entity-relationship like diagram of the data structure on which BRaHS is based is illustrated in figure 5-1.

Figure 5-1: An entity-relationship diagram for data structure used by BRaHS. The boxes are entities and the lines (forks) between them illustrates the relations. A fork means that it is a one-to-many relation. A thick box around an entity indicates that this entity is required in a concrete instance to contain the default values used for classification, the protocol, etc. A dotted box around an entity indicates that this entity is included due to the requirement of local control. The four entities on the right is considered required in order to handle certain needed facilities, which in relation to the concept of coordination mechanisms belong to the $\beta$-level. The structure is explained and discussed further in the text.

The ‘Bug’ entity in the middle of the diagram contains all the information about the registered bugs. For future extensions have we introduced a ‘BugRelation’ that can connect the bugs in pairs. The ‘Classification’ entities contain information on the classification of the individual bugs. Each bug can, of course, have more than one classification related. The ‘Protocols’ entities contain
current and previous protocol for each bug. The ‘Bug’, ‘BugRelation’, ‘Classification’, and ‘Protocols’ are the entities required to contain the required information on each bug. If we think of BRaHS as an active artifact (Simone et al., 1994a) these structures are required for containing the information that are to be mediated by the mechanism (or active artifact), i.e., they contain the conceptualizations of the field of work needed by the actors.

The content of the entities ‘Modules’, ‘Instruments’, ‘Class’, and ‘Level’ are initiated whenever a new version of the mechanism is instantiated, i.e., when a new project is defined (cf. section 3.7). They are defined on the β-level of the mechanism (cf. Simone et al., 1994a), and the content can therefore not be changed during normal use (on the α-level). In opposition to this can the modules and instrument components be grouped and re-grouped at the α-level. In order to handle this have we designed the ‘Mgroups’, ‘IGroups’, ‘ModuleGroups’, and ‘BugIds’ entities.

On the right side of the entity-relationship diagram is shown the entities ‘Connections’, ‘Stages’, ‘Roles’, and ‘Relations’. These contain the protocol as it is described at the β-level. These are only accessible and manipulatable on the β-level of the mechanism. This is not reflected in the interaction with BRaHS. Each ‘Connections’ define a links between two ‘Stages’, which then is related to one or more roles. The ‘Relations’ entity contain information needed in order to be able to establish the required selection lists in the bug registration window. Hence, although this is considered a data structure to be used on the β-level it is very much an implementation dependent structure.

Also the content of the ‘Actors’ entity are defined when a new version of the mechanism is instantiated, i.e., when a new project is defined (cf. section 3.7), but in opposition to ‘Modules’, ‘Instruments’, etc. can these be changed (or new one defined) during the normal use. We have chosen to design BRaHS so that actors can be added or changed on the α-level.

‘AGroups’ is used for keeping the groupings of actors that the users define during the use, i.e., the structure is needed to fulfill the malleability requirement regarding changes to which actors are involved and how they are grouped. ‘ActorsinRoles’ connects the (groups of) roles to each bug. Testers, spec-teams, and designers can thus be related to none, one, or many bugs.

‘Nodes’ and ‘Links’ contain the basic components in the default protocol that were established as an instance of the protocol defined on the β-level (cf. ‘Connections’ and ‘Stages’). Again these can be considered as containing the basic informations on the protocol defined when the mechanism were first established. Similarly to some of the other entities mentioned ‘Nodes’ and ‘Links’ will be initiated when a new mechanism is established (a new project is defined through the ‘new project’ command.

This was only a very brief description of the data structures used in BRaHS. For a more thorough and detailed discussion see Albert (1995).
6. BRaHS considered a coordination mechanism

As mentioned earlier the design and implementation of BRaHS has been driven by the requirements for computer support of coordination of software testing as we have described several places (Carstensen and Sørensen, 1994b; Carstensen et al., 1994; Carstensen et al., 1995b). Although ideas and concepts from the Coordination Mechanisms framework have provided input, the requirement specifications have mainly been based on findings from the field study, and so have the current design of BRaHS. It is thus relevant to discuss how the overall concepts of Coordination Mechanisms fits (or relates to) our design. This section aims at describing BRaHS by means of the central concepts in the framework of Coordination Mechanisms.

6.1 BRaHS as a protocol

A coordination mechanism has been defined in previous COMIC Strand 3 deliverables 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. Furthermore, a computational coordination mechanism is defined as a computer artifact that incorporates aspects of the protocol of a coordination mechanism 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.

Let us elaborate a bit further to what extent we can call BRaHS a computer-based coordination mechanism with respect to each of the essential characteristics for coordination mechanisms defined in COMIC Deliverable 3.3 (Simone and Schmidt, 1994):

1. BRaHS contains an incorporated protocol that stipulate certain aspects of the coordination of the distributed software testing, bug diagnosing, software correction, and verification process. BRaHS encompass a set of prescribed procedures for how the information must be routed from each actor (role) to the next in line. This routing is conducted automatically and reduce thus the complexity of the coordination work in two ways: First, the information is handed over—and the receiver is notified—automatically, and second, the receiver is specified in advance so that the user don’t have to consider who to address the information for. We have however allowed the user to, to some extent, overrule the flow (cf. section 3.5). These situations will require more effort in terms of coordination by the user. It is on the other hand an overall requirement for coordination mechanism that support for local control is provided.

Using BRaHS will require that a set of conventions for its use is established among the actors: The users must use the system. Since BRaHS is
not directly related to the field of work—the software being tested—there are amongst other things no automatic registration of data. Also conventions on how to use the classification scheme, negotiate an estimate, etc. are needed if the system shall be of any use. This type of requirements for use conventions will exist for almost all kinds of artifacts used by more than one actor.

(2) The structures and informations in BRaHS is conveyed by a symbolic artifact. The PC running BRaHS, the code stipulating the flow, and each instance of the database containing information on registered bugs in a specific project are symbolic artifacts. They are thus persistent to changes in the actual field of work accessed, i.e., they can be accessed by the involved actors independently of a particular moment in the work flow and independently of a particular actor.

(3) To elaborate a bit further on characteristics of a symbolic artifact can we say that BRaHS is distinct from the state of the actual work conducted (the field of work). Changes to the state of the testing, diagnosing, correction, and verification work are not automatically reflected in changes the content of the data stored in the databases. And vice versa: Changes to the content of the data stored in the databases made by one of the actors will not automatically be reflected as changes to the code or the work processes.

(4) BRaHS mediates information relevant for handling the required coordination of the distributed activities. First BRaHS mediates information on the registered bugs from one actor to others, either as a notification to a particular actor or upon request. Second, changes to the protocol is conveyed to other actors as notifications (if the originator choose to inform other involved actors, cf. section 3.5), or upon request. These can then, by the actors, be studied in terms of visible changes to the protocol for the particular bug. Since we have decided to let the user decide on whether change information must be send out or not, this implementation will not in all situations explicitly mediate all changes to the state of execution of the protocol to other actors.

(5) All registrations and presentations of information in BRaHS is based on a standardized format for the data and the protocol included. The format can thus be seen as providing affordances to the coordination work since: The pre-specified structure supports the actors in filling in the correct and required information and helps them browse for specific information. The pre-specified structure will also impose constraints on coordination work by not allowing the user to, for example, define his own module names, etc. while describing the diagnose of a bug.

It should be noticed that BRaHS don’t have any validation of the content of entry fields as it is implemented now. Only selection lists can be seen as actually constraining the actors. It will, however, be quite easy to add validation facilities to the entry fields in BRaHS.
Having these characteristics we must definitely, and not surprisingly, characterize BRaHS as an example of a computer-based coordination mechanism.

6.2 The distinction between $\alpha$, $\beta$, and $\gamma$-level

As mentioned earlier the design of BRaHS was not related to the $\alpha$, $\beta$, and $\gamma$-level-architecture in the concept of Coordination Mechanisms (Simone et al., 1994a). It might thus be relevant to see how the layers can be used for describing BRaHS.

Most of the visible actor - BRaHS interaction can be considered as being on the $\alpha$-level. As described for the architecture the $\alpha$-level can be considered a running instance of the mechanism. When a new project is started, a new instance of the database containing information on registered bugs, involved actors, involved modules, work plans in terms of platform periods, instruments, and a protocol for the workflow is generated. This is done by means of an additional setup-application, and is not available in BRaHS. We can thus think of the setup-application as the $\beta$-level offering a set of structures for constructing mechanisms having specific features.

The data structures, their content, and the protocol specified for a particular instance of BRaHS are clearly at the $\alpha$-level of the architecture. The user can interact with these, change the content according to the specified rules, etc. Changes to the setup-application will not affect already instantiated versions of the data structures and protocols. But, since the running code of BRaHS is frequently started up changes made here will, as it is implemented now, influence the performance of the mechanism. This can, of course, be handled through having different versions of the run time code, but this is not implemented at the moment.

Regarding the protocol—and making changes to it—the implementation of BRaHS and the overall ideas in the architecture (cf. Simone et al., 1994a) is comparable. BRaHS allows certain manipulation to the protocol at what would be considered the $\alpha$-level. Only certain changes are accepted. To make more radical changes or to specify a completely new protocol requires access to the setup-application which can be compared with the $\beta$-level. One essential idea in the architecture is however excluded in BRaHS. The notation for the architecture suggests a ‘make permanent’ facility at the $\alpha$-level which can be used to make a specific re-specified protocol into a permanent instance of a $\beta$-level protocol, i.e., a protocol that can be used when instantiating new projects. This facility is not available in BRaHS.

As BRaHS is implemented now it is not possible to identify specific features, ideas, etc. that can be characterized as being on the $\gamma$-level. Some aspects of the entity-relationship structure (cf. figure 5-1) could perhaps be regarded as structures that it would be relevant to have on a very basic level, but we have not aimed at establishing so basic structures when designing BRaHS.
Regarding the database structure it can be seen as containing both structures relevant for the \( \alpha \)-level and structures needed to handle facilities required on the \( \beta \)-level. Some entities were required to contain information on specific registered bugs. These are clearly related to the \( \alpha \)-level. This can also be seen from the fact that they are manipulatable through BRaHS. Other entities were needed for storing information on basic protocol components (e.g., roles and stages). These are obviously related to the \( \beta \)-level, and can only be manipulated via the setup-application. We had, however, to invent a third category of entities. These contained either the default protocol or they were used for grouping actors, modules, and instruments. These were typically instantiated at the \( \beta \)-level via the setup-application, but they were also accessible from BRaHS. Some of them did we also choose to make manipulatable through BRaHS (e.g., groups of actors). This third category of entities are required in order to provide the required selection lists and to provide some facilities for local control. We have called these structures ‘bottom \( \alpha \)-level instances’. Section 5 contains a discussion of the entity-relationship structure used in BRaHS.

A brief conclusion must be that the \( \alpha \)-\( \beta \) distinction suggested in the architecture seems to relevant and useful. Most of the structure we have chosen can be related either to the \( \alpha \)-level or the \( \beta \)-level. There are, however, a number of deviations regarding where to place the functionality in the application. For the database design can we also identify some structures that clearly belongs to either the \( \alpha \)-level or the \( \beta \)-level. Our design seems, however, to indicate that a level “in between” the \( \alpha \)-level and \( \beta \)-level is called for. This must be discussed in relation to a further development of the notation and architecture.

### 6.3 Objects of articulation work reflected in BRaHS

One of the central conceptualizations in the Coordination Mechanisms framework is the ‘articulation work model’ that identifies the elemental objects of articulation for a work situation and identifies basic operations on these objects (cf. Deliverable 3.2 and 3.3).

When the field study was conducted this structure influenced the analysis and the structuring of data. A model for the essential objects of articulation and related operations was defined (reproduced in figure 6-1). When discussing BRaHS in terms of coordination mechanisms it could be interesting to relate to this structure too.
<table>
<thead>
<tr>
<th>Nominal</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objects of articulation work</strong></td>
<td><strong>Operations with respect to objects of articulation work</strong></td>
</tr>
<tr>
<td><strong>Operations with respect to objects of articulation work</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Articulation work with respect to the cooperative work arrangement</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Role</strong></td>
<td><strong>Committed actor</strong></td>
</tr>
<tr>
<td>- Tester</td>
<td>assume, accept, reject [Role]; initiate [Activity];</td>
</tr>
<tr>
<td>- Responsible designer</td>
<td></td>
</tr>
<tr>
<td>- Spec-team member</td>
<td></td>
</tr>
<tr>
<td>- Central file manager</td>
<td></td>
</tr>
<tr>
<td>- Platform master</td>
<td></td>
</tr>
<tr>
<td><strong>Task</strong></td>
<td><strong>Activity</strong></td>
</tr>
<tr>
<td>- Correction task</td>
<td>initiate [Committed actor]; done by [Actor-in-action]; realize [Task]; make publicly;</td>
</tr>
<tr>
<td><strong>Human resource</strong></td>
<td><strong>Actor-in-action</strong></td>
</tr>
<tr>
<td>- James</td>
<td>initiates [Activity]; does [Activity];</td>
</tr>
<tr>
<td>- Jones</td>
<td></td>
</tr>
<tr>
<td><strong>Articulation work with respect to the field of work</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Conceptual structure (conceptualization of field of work)</strong></td>
<td><strong>State of field of work</strong></td>
</tr>
<tr>
<td>- Bug classifications</td>
<td>classify, instantiate; direct attention to, make sense of; act on;</td>
</tr>
<tr>
<td>- Software modulation</td>
<td></td>
</tr>
<tr>
<td>- Platform periods</td>
<td></td>
</tr>
<tr>
<td><strong>Informational resources</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Material resources</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Technical resources</strong></td>
<td><strong>Technical resources-in-use</strong></td>
</tr>
<tr>
<td>- Test machine</td>
<td>categorize;</td>
</tr>
<tr>
<td><strong>Infrastructural resources</strong></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-1: A model of the objects of articulation and related operations for the coordination of bug treatment identified in the field study. Adapted from Carstensen (1994).

It is important to notice, that the model distinguish between nominal and actual objects of articulation. Nominal covers structures having a status so that they can be used, established, committed, etc., i.e., typically the conceptualizations of potential resources and structures used when planning and scheduling the work to be done. Actual contains structures that are involved in ongoing actions or are allocated for specific tasks, obligations, etc., i.e., conceptualizations related to already started actions, committed actors, or reserved resources.
This distinction appears in BRaHS as well. For example in the database structure. Actors and modules can be regarded as nominal structures. They are roles and potential responsible module. But when a relationship structure to specific bug is made (during the diagnosis conducted by the spec-team) the structures become actual as “committed actor” and “module to be corrected”. The distinction is also visible in the user interface of BRaHS. When a user is establishing new actor and module groups, or he is using the selection lists for registering bugs, he is confronted with structures that are nominal in terms of objects of articulation. But, when the user is browsing or searching for specific information on who has which obligations in relation to the next platform period, then he is facing structures that are actual in terms of objects of articulation. There are other similar examples of how the distinction between nominal and actual structures is reflected in BRaHS, but what is more interesting is that, if we had implemented facilities from the related mechanisms handling the software architecture, the human resources, and task scheduling a long list of switching between nominal and actual could have been illustrated. The software architecture and the human resources are only represented as sequential lists in BRaHS, and the task scheduler is not included at all.

If we look at the objects of articulation work identified in the analysis of the field study (cf. figure 6-1) most of the nominal are reflected in BRaHS: Role and Human resource can be recognized immediately in BRaHS. Task is implicitly reflected since all registered and accepted bugs can be considered a task to be conducted. Regarding Conceptual structures the modules and instruments must be considered as such. Informational, material, technical, and infrastructural resources are not included in BRaHS.

Regarding the actual objects of articulation work a few of them is identifiable in the system. Committed actor can be found for all accepted and diagnosed bugs. BRaHS do not register any specific ongoing activities, neither activities related to the testing and correction work itself nor activities related to the coordination. Structures that can be described as Activity and Actor-in-action are thus not included. The same goes for informational, material, technical, and infrastructural resources. The filed information on registered bugs contain certain aspects of the state of the field of work. These can thus be described as actual conceptual structures (cf. figure 6-1).

6.4 Considering BRaHS an active artifact.

In the notation developed for specifying coordination mechanisms the notion of active artifacts plays a central role, cf. chapter 1 in this deliverable and Simone et al. (1994). An active artifact has a “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” (Simone and Schmidt, 1994, p. 96). The mechanism (active artifact) must contain information on, who can have access to its content and which kind of access, and
it must contain information on which other mechanisms it can request for information. Furthermore, the mechanism must include a specification of which triggering conditions it must react to, and what the reaction should be. Such an active artifact has previously been described as demons consisting of an indefinite loop that: First, polls for update or read requests. Second, executes the update requested or returns the requested information. Third, checks if one of the pre-defined triggering conditions are fulfilled, and if this is the case then initiates the actual triggering action (e.g., broadcast information or sends a request to a specific mechanism).

BRaHS has some of the mentioned characteristics. It can be regarded as a mechanism (or application) polling for user actions, and then, when certain pre-specified conditions are fulfilled, trigger a certain action. But it is all included in one application containing only information of its own structures. It can thus not request other mechanisms for information, or trigger actions in other mechanisms. In a situation where, for example, a scheduler was implemented too, it would be obvious that BRaHS should be able to request information from this, or provide information on new tasks to be included in the scheduler. As BRaHS is implemented now, it is organized as an isolated application only triggered or activated through user interaction.

The implementation of the protocol in BRaHS will provide almost the same functionality as if we had designed it as a series of tester, spec-team, designer, platform master, scheduler, etc. active artifacts. The difference appears when we look at the generality of the architecture. BRaHS is designed for a well specified purpose. If we want to expand this we cannot just add an extra active artifact. We have to redesign the whole application. BRaHS is thus much more inflexible in its structure.

The following paper (chapter 7) in this deliverable contains a more elaborate example of how the notion of active artifacts can be used for designing a computer bases coordination mechanism for software bug handling.

7. Requirements of for malleability and linkability

Both COMIC deliverable 3.1 and 3.3 (Simone and Schmidt, 1993; Simone and Schmidt, 1994) have discussed overall requirements for computer-based coordination mechanisms. Based on empirical studies of CSCW systems and analyses of artifactually embedded protocols identified in field studies have we identified a set of overall requirements for computer-based coordination mechanisms regarding: global and lasting changes, local and/or temporary changes, partial definitions, visibility, control of propagation of changes, the reflection of structures of the field of work, linkability, and—for some mechanisms—history.

The previous sections have included several discussions on whether and how these requirements are fulfilled in BRaHS. To complete the picture will we briefly
go through each of the requirements, describe how it is reflected in BRaHS, and discuss potential alternatives.

1. Global and lasting changes:
   A computer-based coordination mechanism must provide facilities for actors to specify and re-specify the behavior of the mechanism.

   This requirement is partly fulfilled in BRaHS. Via the setup-application can the actors (re)specify the module-structure, involved actors, roles, classification schemes, etc. And through BRaHS itself can actors, roles, and modules be grouped and reorganized. There are, however, no means for identifying completely new conceptual structures, for example references to the design specifications used.

   An alternative design would have been to 1) establish a more coherent set of conceptual structures for software testing including for example the software architecture and the platform oriented work plans, 2) design a set of protocol components providing more flexible building blocks for the users designing the protocols, 3) allow the save a running protocol as permanent. These three steps can be compared to some of the basic ideas of the β-level in the notation.

2. Local and/or temporary changes:
   A computer-based coordination mechanism must give actors control of the execution of the mechanism in order to cope with unforeseen contingencies.

   BRaHS allows actors, roles, and modules to be grouped and reorganized during execution, i.e., without need for a new instance. Furthermore can the protocol be redefined individually for each bug registered. It is, however, only changes included in the pre-specified set of deviations that are accepted.

   A redesign of BRaHS should consider how the actors can be supported in redesigning a running protocol in a more flexible manner. It would, for example, be obvious to consider how to allow the actor to connect the nodes arbitrarily. Furthermore would a facility through which the user could selected any nodes in the protocol and restart the process from there improve the flexibility.

3. Partial definitions:
   A computer-based coordination mechanism must allow the actors to let attributes to be left un-specified so that another mechanism can fill it in, or it can be filled in later through actions taken by one of the actors.

   As BRaHS is designed now are there no restrictions on what must be filled in by one actor before he is allowed to pass it on to the next. But there are no features supporting that other mechanisms can infer and specify the un-specified attributes.

   When redesigning BRaHS it must be carefully considered which fields that has to be filled in during registration, which fields that could be
filled in through requests to other mechanisms (applications), and which fields other actors could be requested to fill in.

(4) Visibility:
The behavior (protocol) of the mechanism must be accessible and manipulatable to actors at a proper semantic level of coordination.
The protocol used by BRaHS is visible and accessible to actors in terms of a diagram consisting icons illustrating the involved roles and arrows showing the flow between the roles. The actors can make manipulations to the protocol via these graphical items. It is, however, only certain pre-specified changes to the protocol that are possible to make.

An overall design idea in BRaHS was the idea of illustrating a possible visualization of the protocol. There a many alternatives that must be considered. One important question to think of is whether the roles should be ‘visible’ in the interface. Roles are useful for describing work settings, but actors involved will rarely think of themselves as conducting certain roles.

(5) Propagation of changes:
A computer-based coordination mechanism must give actors means for controlling the propagation of changes to the specification of the behavior of the mechanism.
Since BRaHS is based on a pre-specified set of possible changes to the protocol, and that these can only be made to the part that is not yet executed has the user full control of the propagation of the possible changes. Regarding the propagation of information on changes can users of BRaHS decide to inform all actors involved in the treatment process of a specific bug.
If a more flexible and freely changeable protocol is offered (cf. bullet 2 and 3 above), then the problems of how to ensure consistency and how to let the changes propagate must be completely reconsidered. This has been regarded out of scope in the design of BRaHS.

(6) Reflection of structures of the field of work:
A computer-based coordination mechanism must reflect pertinent features of the field of work.
BRaHS is based on conceptualizations of the field of work and of structures of the work arrangement. These conceptualizations were used by the users themselves when characterizing their work and coordination work activities.
As mentioned in bullet 1 it would be obvious to improve BRaHS so that conceptual structures reflecting to the software architecture, the actors involved, and the work plans are included too.

(7) Linkability:
A coordination mechanism must provide facilities for establishing links to other coordination mechanisms of interaction in the wider organizational context.

BRaHS does not have any (computer-based) linking to other mechanism at the moment. There are, of course, links to the work plan schedule, to the module architecture, to actor lists, and to the directory structure supporting the software integration. These links are however all based on one or more human actors (users of BRaHS) who manually export information from BRaHS to one of the mentioned mechanisms, or manually transfer information from one of the mechanisms to BRaHS.

The very central ideas in coordination mechanisms on linking must be included in BRaHS. This calls for a complete redesign of both the data structures used and the way the functionality is partitioned. Linking has not been explicitly addressed in the current design of BRaHS.

(8) History:

A computer-based coordination mechanism must include features through which the actors can establish an overview of how the information mediated, and the protocol stipulating the flow have evolved over time.

Some of the information registered on a specific bug in BRaHS has explicit references to the originator and the time. For most of the information will it, however, not be possible to see a time or originator stamp. Regarding the protocol includes BRaHS a facility for viewing earlier versions of the protocol.

According to our field study the history information provided by BRaHS is sufficient. But more detailed studies of actual use of a computer-based coordination mechanism supporting testers and designers might point at other historical information needed.

The descriptions of the requirements included above is mainly based on the list on page 22-25 in Schmidt (1994a). For a more detailed discussion of the requirements see this reference or COMIC deliverable 3.1 or 3.3 in general.

8. Conclusion

This paper has described and discussed a prototype of a coordination mechanism supporting testers and designers in coordinating the registration and treatment of software bugs. The design has been based on requirements derived from a field study conducted at Foss Electric A/S.

The aim of this paper has been to illustrate how some of the ideas in the concept of Coordination Mechanisms could be transformed into an actual user interface design. The effort has been spend on user interface design and visualization rather than on designing an example of the structural aspects of the notation.
The prototype, BRaHS, must be regarded as an early horizontal prototype only. At lot of redesign, refinement, etc. must be done in order to develop it into a piece of running code to be used at Foss. It has, on the other hand, illustrated a number of ideas and provoked a lot of discussions and reflections on design of coordination mechanisms. Several of these ideas and problems need further consideration:

Malleability and local control are two essential requirements for coordination mechanisms. The design of BRaHS has raised the question of, to which extent this flexibility must be provided. The spectra spans from complete freedom for the actors to make whatever changes to the work flow they would like to to a completely rigid system allowing no deviations at all.

The notion of roles is important and needs further consideration. Roles are important and essential when characterizing coordination work. But should these roles be explicitly reflected in the user interface? The actors at Foss would recognize the roles of testers, designers, spec-team members, platform master, etc. But when conducting their daily work they would not necessarily be aware of whether they were working as designer or platform master in a specific case, and they would be constantly switching between the roles. We can on the other hand not only use the actors as actors when conceptualizing the human resources. This would imply to in-flexible systems. Thus, both the use of roles and their appearance in the user interface of coordination mechanisms must be considered further.

It appeared to be fruitful to discuss the application in terms of $\alpha$-level and $\beta$-level (it was not relevant to discuss the $\gamma$-level in relation to BRaHS), especially when considering where to place specific facilities in the user interface. The advantages was that it provoked a more explicit consideration with respect to these facilities, cf. section 6.2. But as “design guidelines” the $\alpha$-$\beta$ distinction did not provide much support.

The design of the basic database was also both supported and confused by the $\alpha$-$\beta$ distinction. The distinction supported the identification of certain structures required at the $\alpha$-level, respectively the $\beta$-level. The problem occurred in relation to the $\beta$-level structures that had to be accessible and visible in the user interface. To deal with this did we have to introduce an extra set of entities.

The concept of objects of articulation work was very useful in relation to which structures to conceptualize. It must be considered further, how to decide on which structures to include and which structures to subscribe to from other mechanisms when designing a specific mechanism.

Visualization of the content of a mechanism and the related protocol appeared, of course, to be a very complex task. The number of possibilities is indefinite, and there is a long series of trade-offs to relate to. Existing literature on visualization must be related to studies of characteristics of coordination work. As an example can be mentioned the monitoring activities involved in coordination work. We
have not, in the design of BRaHS, carefully considered which information the actors need to overview, and how this could be presented.

During and after the design of BRaHS were there several basic questions related to design of coordination mechanisms that were left open. The most important of these were firstly, how do we decide how to propagate changes to a running protocol and how do we “inform” the rest of the system? Secondly, how do we handle concurrency? It is not a problem to process many bugs concurrently, but we have no support for handling situations where the processing of one bug influences how another bug should be processed. This will often be the case in complex coordination. And thirdly, the question of how to organize linking between mechanisms? BRaHS was very traditionally designed, and did not benefit from the ideas of linking between coordination mechanisms.

The latter problem mentioned above can perhaps be generalized into a general problem that might be worth discussing: How can we use the conceptual framework of Coordination Mechanisms, the notion of the $\alpha$-$\beta$-$\gamma$-level notation, and the overall requirements for coordination mechanisms when designing traditional applications to be used for supporting the cooperative aspects work?

This concluding section has raised a number of ideas, problems, and questions, rather than solving or answering any. To inform discussions on the above listed ideas, problems, and questions future work must include several things: Development of new and more elaborate prototypes of coordination mechanisms are required. The use of these prototypes must be studied in actual real-life work settings. And finally the conceptual framework of Coordination Mechanisms directing which aspects of the cooperative work to address must be expanded, refined, and improved.

Acknowledgments

Thanks to numerous people at Foss Electric. This research could not have been conducted without the invaluable help from them. Thanks to Kjeld Schmidt, Carsten Sørensen, and Tuomo Tuikka for many fruitful discussions of the analysis of the field study, and a special thanks to Liam Bannon for pinpointing a number of relevant aspects during our evaluation sessions. All poor design decisions and errors in this paper naturally remain the responsibility of the authors.

References

Albert, Thomas (1995): BRaHS. En EDB-baseret Koordinationsmekanisme til håndtering af Softwarefejl [BRaHS: A computer based coordination mechanism supporting the handling of software bugs], Computer Science Department, University of Copenhagen.


Chapter 7

Architectural Issues in Design of Computational Coordination Mechanisms for Software Testing

Tuomo Tuikka° and Carsten Sørensen•

°University of Oulu, Oulu, Finland
•Risø National Laboratory, Roskilde, Denmark

Previous chapters in this deliverable have reported from the analysis and design and implementation process aimed at demonstrating the concepts of computational coordination mechanisms (C2M) within the field of coordination of distributed software testing. This chapter documents the development of a concept demonstrator focusing on architectural issues of providing computational coordination mechanism functionality supporting the coordination of distributed software testing through linking of active artifacts. We will in this chapter demonstrate the protocol of a set of C2M’s in an active artifact, the objects of articulation work in an example of notifications, malleability and linkability of the C2M’s. The application and its structure is, furthermore, related to the layered structure of the notation. Finally, we will reflect on the chosen architecture and on future developments by assessing the possibilities of using Apple Open Collaborative Environment and OpenDoc as a platform both for integrating computational coordination mechanisms, for providing malleability at the end-user level, and for providing flexible ways of linking C2M’s.

1. Introduction

For the past 3 years research activities in Strand 3 of the COMIC project have produced a conceptual foundation for coordination mechanisms, a set of requirements for coordination mechanisms allowing identification of specific characteristics of the notation for controlling and specifying the mechanisms, and an initial definition of the notation.

The domain selected for demonstrating the concepts related to computational coordination mechanisms is coordination of distributed software testing, or more precisely, the coordination of distributed registration, classification, diagnosis and verification of software errors as analysed in the Foss Electric case study. Previous research on this topic, as documented in previous chapters in this deliverable and in Deliverable 3.2 and 3.3, forms the foundation for this chapter (See for example: Carstensen, 1994; Carstensen & Sørensen, 1994; Carstensen et al., 1995b; Carstensen et al., 1995a).
The aim of this chapter is to demonstrate and discuss the concept of computational coordination mechanisms (C2M’s) by presenting a computer-based concept demonstrator and by discussing the possibilities to add features of the concept to an operating system platform.

Gordion, the demonstrator, serves primarily the purpose of providing concrete feedback to the research by providing knowledge about present technologies to support the implementation of a system of C2M’s. Central concepts related to computational coordination mechanisms are demonstrated and discussed. These are primarily: Malleability and linking of C2M’s, the notion of active artifacts, notifications and their behavior according to the selected protocol, and the architecture based on the notation. The last of these concepts is presented as a layered and modular structure of the notation in comparison with the application design and the software architecture of the concept demonstrator.

The development experiment provided important experiences on how to design and implement the architecture for a computational coordination mechanism based on rich field study data. The implemented architecture supports an unlimited number of instances of each of the dedicated mechanisms to be connected in a local area network. Experimenting with an architecture consisting of a set of applications dedicated to support the various roles and being based on bug forms as documents rather that fields in a central database, provided us with a sound foundation for moving to the next step of embedding the functionality in OpenDoc components. Working on the architecture also gave inspiration to the idea for further development of providing overview of the protocol by agents inspecting current status of each active application.

Section 2 outlines the research problem addressed in this chapter. Section 3 presents the requirements for the concept demonstrator. Section 4 provides an overview of the Gordion demonstrator. Section 5 relates the demonstrator to the concept of active artifacts. Section 6 discusses malleability and linking issues related to the demonstrator. Section 7 reflects on the architectural issues related to developing the demonstrator. Section 8 concludes this chapter with reflections on what we have learned from developing the demonstrator and with a discussion of possibilities and limitations of extending a computer system with computational coordination.

2. Research problem and context

A constructive research approach is one way of experimenting with the application of concepts and theories. It provides the possibility of presenting concrete manifestations for further discussion and critique. At this stage of the research on computational coordination mechanisms, a constructive approach was seen as a suitable means for conforming a starting point for continuing development and application of the notation.
The aim of this chapter is, by means of the Gordion concept demonstrator, to present and discuss linking and malleability of computational coordination mechanisms, and to discuss architectural issues pertaining to developments in state-of-the-art within component software and the Apple operating system. The field study at Foss Electric analyzed and modeled a bug handling mechanism which is used in software testing work (Carstensen, 1994; Carstensen et al., 1995b; Carstensen et al., 1995a). This mechanism can be considered essential in software testing to register and route the bug information. The new design suggested for a computerized version has been selected as a demonstration example (Carstensen & Sørensen, 1994). Hence, a concept demonstrator was built, prototyping the features of a computational coordination mechanism. The work documented in this chapter was conducted in parallel, and coordinated with, the development of the demonstrator presented in the previous chapter. Because user-interface issues have not been a main issue in this work, the Gordion demonstrator only reflects incremental changes to the lay-out specified by Carstensen & Sørensen (1994).

The development of concept demonstrators is part of the effort to bridge the gap between on the one hand, the field study of coordination in software testing at Foss Electric, and on the other hand, the development of the notation for computational coordination mechanisms. Figure 1 shows the conducted and planned stages related to the field study, analysis of results, design and implementation of demonstrators, operating systems issues, and feed-back from software developers at Foss Electric. This chapter is represented in the figure as stage 5. The previous three chapters reflect stage 1–4, and the following chapter documents work on stage 6. Stages 7 and 8 denotes future work.

![Figure 1: The main stages of the research in computational coordination mechanisms for distributed software testing, based on the Foss Electric field study of coordination mechanisms in software testing.](image-url)
3. Background and requirements

This chapter briefly outlines the background of the bug handling coordination mechanism and the requirements for the demonstrator application. The bug handling coordination mechanism is adapted from a case study at Foss Electric and will from now on be abbreviated as the Bug CM (Coordination Mechanism) and its computerized version as Bug C2M (Computerized Coordination Mechanism).

3.1 Design of the Bug C2M

A bug handling report in software testing is a method for documenting a software error and informing them to other people in the software development ensemble. Thus, it is an essential part of software testing work and reporting the problems (Kaner, 1993) as seen in the Foss Electric case (Carstensen, 1994).

Since computerized systems offer possibilities not present in manual systems, the original artifact was made subject to an initial redesign (Carstensen & Sørensen, 1994). This report does not discuss this new design, but uses it as a basis for building a user interface for the demonstration. For a more elaborate discussion on user-interface matters see Chapter 6 in this deliverable.

![Figure 2: A possible UI-design of the demonstrator and a redesign of the physical bug form to serve implementation of Bug C2M.](image)

The protocol of the Bug C2M is also derived from the field study, where 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. Several groups of actors are involved in this process. They are, thus, users of the form and of the procedures. These actors are assuming the roles of: Software testers, specification team members, designers and platform master. A 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.

3.2 Requirements for the demonstrator

The final Bug C2M system will be integrated into an operating system environment, using available features of the operating system and supporting linking between document handling systems, project management systems, spreadsheet applications, databases etc. The demonstrator is a set of stand-alone applications which illustrates some concepts of C2M’s. We do, therefore, not to consider, for example, how the distributed registration of the bugs will be performed and in which environment, but rather demonstrate this with some functionality using several workstations.

The overall requirements chosen for this demonstrator experiment focus on the malleability and linking of CM’s. Three C2M’s from the field of software testing are implemented; the bug report, the project schedule and the binder. It should be possible to specify a C2M by means of a ‘fixed grammar’ presenting aspects of the notation. Finally, a there will be a demonstration of linking to other CM’s in the field, so software architecture and human resources should be presented as mock-ups.

The demonstration will not focus to user interface issues—the idea is to discuss the architectural issues in the design of C2M’s. The demonstrator will show an example of the protocol routing the CM through different roles of the coordination process. The roles would be one example of the objects of articulation work. Because of the chosen architecture, only the foundation for visibility of the protocol has been implemented. In further development of the demonstrator, this issue will be covered.

4. The Gordion demonstrator

Gordion is the name of the group of demonstrators which we have implemented. The architecture of dividing the bug form mechanism into several applications, has also been the motivation for the name ‘Gordion’. Instead of building one application providing support for all roles, we decided to divide it into pieces which are presenting the role of the user. This is similar to the famous way of opening a knot with a sword and splitting it into two halves. Thus, the overall design of the demonstrator fulfills easily one of the requirements, i.e., presentation CM’s in separate workstations. There will be more thorough discussions of this design decision in later chapters.

The Gordion demonstrator manifests itself in a number of applications each dedicated to support one role in the cooperative work arrangement. Each application can be executed from the Apple OS Finder desktop, and can store documents in the files system. Figure 3 illustrates three Gordion demonstrators
with documents—the next version will include separate icons for each application.

Figure 3: The Gordion demonstrator resides as separate applications on the Apple OS Finder desktop.

In Figure 4 we can see an exemplary principle of implementation of the demonstrator application divided according to the roles in different machines. The example shows three workstations, but several instances of each of the applications can be executed and linked on the same workstation. There are, furthermore, not implemented limitations to the number of mechanisms being linked.
Figure 4: Three workstations with Gordian linked to each other through a network. Workstation 1 contains the application of including the protocol needed for software tester role. Workstation 2 is dedicated to the spec-team role, thus anyone having a spec-team role application would be able to do tasks specific to that role. Finally, workstation 3 is executing the project schedule, which is used by the platform master. Schema of the principle of the demonstrator.

A basic example of the demonstrator in use could be the routing of the Bug CM presented in the previous chapter. When the software tester finishes his or her registration, the information will be routed to the connected mechanism(s). The applications are connected by specific linking addresses to a different workstation. Linking of the demonstrators and linking between the CM’s are not the same concepts, since linking between CM’s means the interoperability between them and CM’s can be created from other CM’s. The linking of demonstrators, however, implies that the applications in different workstations are linked also between the same CM and the addresses of the workstations are defined explicitly by the users.

One basic feature of the demonstrators is the possibility to have many Bug C2M documents running at the same time. The tester can, for example, bring up many Bug C2M’s on the desktop, swap and browse them modelessly even after the Bug C2M has reached later stages in its life cycle, and bring up the history of the specific bug continuously to the testers Bug C2M. Although the bug forms can be swapped, all the services for all of the instances will be available. Thus, they can be saved and retrieved as individual bug documents. This is the overall functionality of one application.

The communication functionality of the Gordian applications is embedded into the general user interface. Different menu selections are used when the user wants to do something which is of some concern to other users. These activities might not even be explicitly considered as communication, but rather as part of the normal work procedures.

A closer look at the messaging techniques between the applications reveals the benefits of the chosen approach of distribution. The messaging techniques which
handles the communication of the applications is called Apple-events. These events can be defined by the developer and they form a messaging protocol inside the applications, thus making the demonstrator aware of the events sent by other applications or by other CM’s. Not only messages are possible, but also queries of different kinds, depending on the role of the user.

![Diagram](image)

Figure 5: The messaging techniques forms the basis for the protocol. Number 1 is a basic message to another application, and 2 presents a query, i.e., a message which returns a result.

The figure 5 shows the principle of sending a message to another workstation (1) and a sending a request to another workstation and getting a response accordingly. All the messages contain identification information in order to be able to dispatch the information as a message, request, notification or just data about the instance. In the example, the communication occurs between two applications of the same CM. An instance of the Bug C2M is the one specific to one bug, however Bug C2M’s in the figure are of the same Bug C2M’s, but not the same instance since they are in different applications. This could be confusing, but remembering that we are using different applications implementing the functionality for the different roles might help a bit.

While the applications are basically independent, the application linking must be defined by someone, which, as previously noted, is a different concept than the linking between other C2M’s. An example of this is that the user must choose the workstations and applications which will be connected according to his role. Figure 6 shows the functionality to do this.
Figure 6: The dialog box supporting choosing the other participants from the network.

The dialog box in figure 6 shows local area network zones as well as all the available workstations in the chosen network zone, thus we can see that the connection also can be established to an application running on the users own workstation. The programs in the right box show that bug reporting applications are available at the workstation Tuomo 700, where BugsSPT, BugsTST, BugsProSche are the active applications. Now we can connect to one of those depending on the role we have—information which principally could be given only once and saved for later use, a feature not implemented in the current demonstrator. A large amount of network applications could be set up even so that one machine could hold all the roles, but this would in the early stage of the demonstrator result to a complex system to handle. Furthermore, more discussion about additional facilities like aggregations and agents to produce them and the feasibility of these facilities is can be found later in this chapter.

The selected hardware is an Apple Macintosh platform since the networking is a built-in feature and the platform provides a transparent way of accessing the network via high level programming languages. Furthermore, software tools allow us a relatively easy way to build user interfaces on top of a powerful programming environment. The software tools used are MetroWerks Codewarrior 6 C++ and the DynaWare™ interface tool, which provides a stable user interface prototyping environment. Furthermore, the architecture and selected techniques of the demonstrator design provide a degree of freedom for the distributed application. Thus, when the protocol is finalized to some role by programming we can principally make indefinite amount of copies of the application and set them to execute functions depending on a specific role.
5. Active C2M for Software Testing

In software testing work at Foss Electric, the Bug CM is a manual system developed by the software development team. It includes required aspects for reporting and handling of bugs. Thus, according to the definition of the CM it actually is a protocol, encompassing a set of explicit conventions and prescribed procedures and supported by a symbolic artifact with a standardized format. The form stipulates and mediates coordination activities and thus reduces the complexity of coordinating the work. The computer artifact (C2M), however, incorporates aspects of the protocol of a CM 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. This chapter discusses and presents the CM as a protocol by explaining what is an active artifact and how it behaves according to a protocol which has been built into the demonstrator. We will also take a further look at the roles as an example of objects of articulation work within the presentation of active artifact and with the discussion on notifications.

5.1 Active artifact

The concept of an active artifact stems from the definition of the C2M, which requires that such artifacts play an active role in the articulation work. The artifact built in the demonstration, the Bug C2M, incorporates properties which allow it to be aware of the environment and interaction with other C2M’s in a restricted way. For example, the artifact recognizes some information types while updating and it reacts to some triggers defined in advance. To be more exact about what active artifact includes we represent the Bug CM in the format promoted by Simone & Schmidt (1995) (See Figure 7).
<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.

Figure 7: The Bug C2M characterized as a frame following the notation from (Simone & Schmidt, 1995).

This structure represents and collects information about the active-artifact Bug C2M. At the same time it also represents a protocol, where one of the determining items is in the first slot. ACCESS contains information about who is allowed to access this form and how—updating or only reading. The trigger defines which events cause an action with the artifact with triggering-condition: update(tester), triggering-mode:spec-team, trigger: send(BR). The rest is a list of information the bug report includes. A computational artifact which stipulates and mediates the articulation work would in this case include also some data structures, like suggested in the previous structure. As we can see they are starting from initials of the actor, instrument; a technical resource, etc.

Now, given that the C2M’s should be 'active' in relation to the users and their computational environment, they should be active in the sense that 'changes to the state of the mechanism induced by one actor can be automatically conveyed'. Activity of the C2M is possible by its ability to identify certain events and information. This feature is attainable with standard 'demons' associated to it. Accordingly, these 'demons' are monitoring the environment in order to detect update/read requests: if there is one suitable available, the update/request is performed.

We can illustrate this with an example. First, to start a reporting and documenting of an error in a software application, the tester must choose to register a bug from the applications menu. This opens—or more specifically instantiates—a
new artifact with all the information shown in the user interface. The tester fills in all the necessary fields so that the bug can be identified and analysed in further stages of the routing. Triggering some event depends on the definition of the underlying protocol. If some of the fields is defined to trigger, a procedure then it is identified and executed.

When the tester is done reporting, he or she can register the bug by selecting a function from menu. In the demonstrator this triggers checks for existence of necessary information. The protocol checks whether or not the required fields are filled in before sending the information further on. If mandatory fields have not been filled in, a notification appears. Figure 8 shows a dialog which pops up when a tester has not filled in the mandatory fields. This is our first example of the protocol which acts on a triggering condition.

![Figure 8: Bug C2M after the check in the protocol where mandatory fields are not filled in.](image)

Principally, the checking procedure of a 'demon' would be continuous during the Bug C2M lifetime, that is, from the situation when the tester adds the first information to the last point of Bug C2M execution. As in the example, the tester-role has access to the artifact and updates the required fields, then the demon examines the values given in the background and the protocol reacts to them accordingly. Thus, although corresponding idea of using 'demons' is not explicitly implemented here, the behaviour of the demonstrator manifests this feature. In fact, the awareness of the environment and triggers is implemented in the application code.

The rules for the behaviour of the active artifact are predefined at some stage of the systems development. Furthermore, these rules can be changed at some degree
freedom depending on the role of the user. This is further discussed in the sections on local control, malleability and the general architecture.

Another important feature of the active artifact is the automatic routing of the bug according to the workflow, thus conveying the artifact automatically to a predefined destination and notifying the necessary and defined partners about availability of new information. The roles shown for access in the previously presented structure define a set of people able to use the mechanism. Consequently, the Bug C2M routing of the artifact will be automatically conveyed between these roles. Predefined and automatic routing address of the demonstration is visible in Figure 9.

As we can see in Figure 9, the routing is according to the protocol set as default to the next step which is the role of specification team and the bug information has been filled by the tester. Thus, the bug information will be delivered to "whom it may concern". The process of routing follows the registration-diagnosis-correction-verification phases visible in the figure.

Finally, the Bug C2M is active not only towards its own properties as explained, but also with other C2M’s. The functionality of linking to other mechanisms and notifying the environment will be discussed in more details in the following sections.
5.2 Notifications

According the requirements, the system should be able to provide features for informing the participants of the state-of-affairs. Also, 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 C2M. Thus, notifications play an important role in the life of a C2M.

There are several situations in which a notification should be used for informing users. Besides notifying a single actor also a group of actors could be notified. According to the interactive nature of the C2M, notifications are sent automatically after a triggering event. Triggering conditions could for example be, temporal, changes of data or modification of the protocol.

We will present examples of how notifications are triggered and scattered around the cooperative ensemble depending on the chosen objects of articulation work which are defined to the protocol for this specific occasion.

For example, it is possible that the C2M notifies all the actors with a role in the spec-team. A simple user interface format of the notification could be as in the following figure. However, the text could change depending on the message, and further functionality in the form of further queries could be implemented.

![Figure 10: A new bug has arrived from the tester, leading to a spec-team member being notified.](image)

In Figure 10 the spec-team member receives a notification about the registration of a new bug. The receiver of the notification could depend on the responsibilities the member has.
Another example could be a change to the protocol. This would also bring up a notification to one or possibly to a group of users. Overriding the default protocol by skipping a role in the routing could result in a notification to the role who should get the bug next for inspection. This actor actually could be anyone who is assigned by the tester in the specific situation, thus C2M would support any kind of exception to the routing. Figure 10 shows the routing to the designer in the bottom left corner, but the pop up menu could include all the necessary organizational information varying from roles to actors.

These examples indicate that the propagation of the notifications depend on the type of the notification and on the specified behavior according to the role.

Different kinds of notifications could exist depending on the settings of the notification type. In the demonstration we can see that the notification is in a modeless window. Thus, allowing us to change from the notification to the Bug C2M or even to the next notification. However, a modal notification could be possible thus being less obtrusive to the work of the spec-team member.

The demonstrator is an example of an embedded mode of indicating that a new bug has been added to the registration: it beeps. Therefore, after the beep is heard the actor can choose whether or not to check the bug information. There could also be a flag indicating this event symbolically. In extremely important cases the C2M could be persistent in getting attention with even different degrees of notification thus indicating the time dependence of the problem. A simple extension of the notification functionality could mimic the notification levels in email clients such as Eudora, where the user configures settings for the "obtrusiveness" of notifications by combining sound, modal and modeless notification.

Finally, different kinds of facilities for handling the notifications could be useful. For instance, there could be facilities available for creating aggregations of notifications in a specified order. In case of a flux of notifications a personal filtering would be useful to restrict them to present necessary information only.
6. Malleability and linking of C2Ms

The concepts of malleability and linking are fundamental and basic requirements to the C2M's. An application which is implemented this way would have functionalities that are useful for changing the predefined protocol not only temporarily, but also permanently. This chapter will present some features which manifest the malleability of a C2M and some features which are used with the demonstrator thus pointing out the requirements for further development. We will present examples of linking C2Ms. The concepts will be illustrated with examples from the demonstrator.

6.1 Malleability

A C2M can be used in a routine-like fashion if changes to the C2M behaviour are not deemed necessary by the user, i.e., changes to the predefined protocol. However, in some situations, due to the distributed and changing nature of cooperative work, users need a freedom to modify their C2M. Temporary change of behaviour can be supported by providing the user with a certain degree of local control of the mechanism. Exercising local control, users are able to define some basic changes to the C2M while they are working with it. In other words the notation would support the manipulation of the C2M 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.

An example of local control is a situation where the tester registers a bug, but the bug is simple to classify and fix, and the problem can be addressed to a specific designer. The spec-team team can then be skipped in the routing scheme. In this case the tester can change the routing of the bug form, overriding the Bug C2M by choosing designer from the routing field —assuming it is known which software module is handled at the moment and that the responsible designer can be tracked through this information. Thus, the responsible designer will receive the bug information. Consequently, in other circumstances different persons could be chosen to be listed as receivers as well. For example, a list of all the actors could be available from the menu shown in Figure 11.
Changes to the behaviour of the C2M could also mean that the organization of the work has been changed. If this is the case, all the future bug handling mechanisms would need to be changed so that they follow the new coordination procedure. A change to the routing of the C2M can then be done by making it permanent by choosing the 'save change' from the main menu. This could be the way to modify the general protocol of one project, while all the other involved in the same project would receive a notification that the protocol has been changed.
The bugs-menu in Figure 12 is only for demonstration purposes, that is why it has some reserved menu items for later use. Notice that item Next New Registration is grayed since we are now using the C2M of the tester. The item could even be missing from the testers user interface.

A far more profound way of malleability is the possibility to change the protocols to meet the additional requirements in a software testing situation. We had already a small example of this. Figure 12 also shows a menu item (save protocol), which could be used for saving the overall protocol for all of the subsequent Bug C2Ms. Building and changing protocols requires, apart from easy-to-use computational support, also user skills depending on the depth and difficulty of the change.

Until now we have shown examples with some simple degree of freedom and when the user can do changes to the behaviour of their Bug C2M. The demonstrator allows us also to do two additional things: we can modify the user interface as much as we want in a separate role and we can add new elements to the existing user interface. These modifications are primarily interesting from the perspective of prototyping the demonstrator, and for experimenting with functionality needed for tailoring the user interface.

Changes to the user interface is possible because it is implemented using the DynaWare™ toolkit which is connected to the recent demonstrator applications online. Thus, we can for instance change the style of the field, their properties, add buttons or fields, change the properties of existing windows. However, if functionality should be added to any of the user interface items it should be constructed in the programming level. The positioning or the properties of the
fields are independent in each application thus providing much flexibility with changes to the UI. We could have added an editor which would have had more malleability to our own changes than a commercial editor. But considering the amount of time and effort we think that this example is rather useful in pinpointing the future requirements from these facilities. The advantage of using the DynaWare toolkit is that all user interface elements are "event-aware". The toolkit defines a layer between the application and the operating system routines, and the code is stored as resources. New fields without functionality can be added and tested on run-time. Although they are not linked to application functionality — this requires recompilation — they are stable because all DynaWare interface objects automatically protect undefined events from the lower layers. A "Registration" radio button, as seen in Figure 12, contains the basic functionality of toggling between on and off when selected. If the toggle event is not intercepted in the C++ program, nothing except the toggling will happen.

It is possible to add new elements to the demonstration, e.g., new roles, actors, software modules or even new objects of articulation work if necessary. However, these changes must be made using a commercial editor, ResEdit™, which is used for changing the resources of a program. Again, the changes will not be reflected to other application so a mechanism for tracking those would be necessary in later versions. This kind of approach to change or add to the objects of articulation work could be one possibility. Namely, an editor which can be used as a level between the user and the places where necessary updates would be compulsory. The other way to do it could be an embedded approach where the 'editor' could be inside the selections of the user but depending on the role of the user. We could then, for example, get different menu selections having all the objects of articulation work like task, role, actors, etc. Then we could have submenus of commands add, delete, and finally a selection to point out a task to someone or make a role responsible of some resource. Finally, there could be a way to attach the object to a user interface object to finalize the change. As noted above, this kind of action demands more user skills than the malleability presented earlier and could be pointed out to a user of certain task in an organization like systems analysis and design. All the further applications or C2M's would then follow this design, thus inheriting all the features from the original application.

6.2 Linked mechanisms

The requirements for C2M prescribe that computational coordination mechanisms must provide means for establishing links with other C2Ms embedded in other applications. Thus, the Bug C2M must have access to other computer based mechanisms in order to provide the facilities required in (Simone & Schmidt, 1995). Figure 13 illustrates the aggregation of entities into C2M's, as analyzed by Carstensen & Sørensen (Carstensen & Sørensen, 1994). The diagram identifies the basic requirements for support of linking among the mechanisms. The
illustration contains only one of several possible decompositions of the mechanisms, and

Figure 13: An entity-relationship illustration of four computer based mechanisms that, altogether, could support the articulation of software testing (Carstensen & Sørensen, 1994). Each of the gray boxes represent a mechanism. The mechanism described in this chapter is the Bug C2M in the center. The forks in the relations indicate a many-relation.
Tasks

Roles

Actors

Registered bugs

Bug categories

Software modules

Figure 14: Entity-relationship illustration of the entities covered by the Gordion demonstrator prototypes. It is important to note that although the demonstrators cover the entities, they only provide a subset of the intended functionality of the C2M's in Figure 13. BugSPT and BUGProScheme basically contain the same data elements, but the former supports individual bug forms, whereas the latter provides an aggregated view.

Figure 14 shows how the Gordion demonstrators cover the same set of entities. The C2M's identified in Figure 13 is intended to provide a broad host of functionality not currently implemented in the demonstrator prototypes.

We can see examples of linking, connections to the other C2Ms, also in our demonstration of the Bug C2M, especially when it is in use by the spec-team. The intention is not to show how the means to establishing links works, but more to the idea how the other C2Ms are visible in our Bug C2M.

During their work, the spec-team accepts and classifies the bug. Also, the platform period field are filled in and the responsible designer is assigned. These fields are linked to the planning C2M and software architecture C2M. The platform period in which the bug is going to be fixed can be set by default depending on the time on the planning C2M, or by selecting the proper period from the attached list (See Figure 15).
Subsequently, the software module involved is tentatively identified and selected from a menu which is linked to the software architecture (Figure 16). Although the list in Figure 16 is not long, one could have a presentation of the whole architecture, maybe in dynamically updated hierarchic menus being constructed, for example, by links to a CASE repository or to the programming environment directory. When the module is selected we already know the responsible designer(s) since the software architecture C2M is connected to human resources C2M (Figure 13 and 14). Then, the C2M selects and fills in the field automatically.
Figure 16: Modules field has a link to software architecture through a menu. The responsible designer is automatically filled according information in software architecture C2M and human resources C2M.

Specification team member can estimate the correction time according to his or her knowledge of the difficulty of the task. The estimated time will be reflected in the platform masters project schedule in the proper position depending on the task and the software module and who is responsible on it. Since the C2M's are actively linked, the platform master continuously can obtain awareness of the current project status. Finally, the bug is filed as accepted and updated in a central database, the responsible designer (and the tester) are notified. When selecting the radio button 'accepted' we must naturally confirm the change from the user that the changes are final before interfering the others work.
C2Ms can be composed of two or more existing ones by linking interface elements. The interface establishes which information can be mutually accessed and communicated by the agents constituting the involved mechanisms. The project schedule C2M needs information about the correction time estimate and it is a composition of information from instantiations of bug handling mechanisms where the definitions determine the position of updates.

A project schedule as one of the mentioned C2M's (Carstensen & Sørensen, 1994) would include at least the following items: task, relevant software modules and actors. The schedule shows how much time has been estimated for tasks to be performed by each connected designer. Furthermore, the project schedule C2M must include at least connections to the software architecture through reference number referring to specification and to the planning C2M through platform period reference.
Figure 18: The Gordion Project Scheduler application.
7. The architecture of the notation and of the demonstrator

Until now we have shown examples of the Bug C2M demonstrator. In the following we discuss the relationship between the demonstrator and the architecture based on a notation of coordination mechanisms. There will first be a short presentation of the different levels in the notation. Then we will discuss how these levels are manifested in the demonstrator.

The architecture specified in (Simone & Schmidt, 1995) identifies different degrees of malleability. The $\alpha$, $\beta$ and $\gamma$-level is suggested as an appropriate architecture, as shown in Figure 19.

In terms of the Gordion demonstrator, let us consider the $\alpha$-level. The user is starting the application and controlling it through the user interface. To open a new bug form, the user opens a new document by choosing a menu item. Consequently, the Bug C2M is initiated, an instance created and the user has access to the protocol. At different levels as conceptualized in the figure, the instance consists of the information of how the mechanism behaves and what information is included. This instance is thus a product defined by a system analyst, designer and programmer in $\beta$-level and $\gamma$-level, but instantiated and used by a user at $\alpha$-level.

Generally, everything the end-user does and accesses can be considered as part of the $\alpha$-level. The field of work appears in the fields of the user interface. Because we are handling bug information, the attributes reflect the information of this object in the field of work. Thus, the end-users can fill in the fields with attribute values depending on the bug. Furthermore, the appearance of the Bug C2M can vary in different roles, thus showing the needed information and reducing the redundant clutter in the UI.
As previously discussed, regarding malleability, the user can modify the instance locally by changing its behaviour by enforcing the change to the routing or changing its appearance. At the $\alpha$-level we could also be able to follow the history of the bug, and changes to the protocol could also be made permanent (currently only implemented as mock-up functionality). Therefore, the protocol can be accessed in several ways depending on the depth of its use.

As user skills and roles change, there will be increased needs and options for tailoring the C2M. Thus, at $\beta$-level the user could be systems designer or systems manager who will be able to define or change the way the C2M works in a more deeper level, thus affecting the way coordination work is supported.

Referring to Figure 19, the $\beta$-level user has access to the grammars constructed at $\gamma$-level. The selections of structures and object types are derived from the field of work. As we explained in the previous section, new roles, actors, tasks etc. can be added with the tools available to build the demonstrator depending on what kind of work situation we want to support. The concept is actually tool
independent in the sense that we can build a user interface in many ways to access, define and change these objects and access the grammar which has been defined somewhere by a programmer.

<table>
<thead>
<tr>
<th>ATTRIBUTE-NAME</th>
<th>ATTRIBUTE-TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>senders</td>
<td>SPEC-TEAM</td>
</tr>
<tr>
<td>content</td>
<td>opinion-request</td>
</tr>
<tr>
<td>receivers</td>
<td>Designer</td>
</tr>
<tr>
<td>IP</td>
<td>request</td>
</tr>
<tr>
<td>preconditions</td>
<td>NIL</td>
</tr>
<tr>
<td>answer-time</td>
<td>2 Days</td>
</tr>
<tr>
<td>awareness</td>
<td>NIL</td>
</tr>
</tbody>
</table>

Figure 20. An example of Interaction at the β-level.

Figure 20 shows an example of interaction specified at β-level. Interaction is considered to be part of an activity OAW, where interaction means exchange of information among interlocutors. Another action type is actions transforming resources and it is called a performative action.

In the demonstrator this would mean the specification of the C2M to have a feature which for example, for the spec-team role would perform opinion-requests to the designer role. This implies that the C2M must contain information about the role or roles it supports. Gordion has a separate application for the specification team, so the protocol is by definition role-aware. Consequently, Gordion should at the α-level have a way to perform the request specified, in one of the menus perhaps and regarding the β-level we have to propagate this feature to the code. Thus, as an intermediary step it would be helpful to build the protocol using a protocol editor or a structure of menus which has all the possibilities included, derived from γ-level. The basic elements in the notation are formal structures and the objects of articulation work. Malleability at the γ-level would mean the possibility to change a grammar or build a new one. Principally, that is what we are doing when we build the application.

The Gordion demonstrator is built in C++, and we have experimented with object class inheritance as a means of expressing the α and β-level. This work is by no means completed yet, but we have learned valuable lessons for the further work.

The Gordion demonstrator applications are compiled instantiations of the class 'BugDocument' which is a skeleton for Apple Macintosh applications which save documents to and open documents from the Finder desktop (see Figure 21).
Figure 21: The object classes of the Gordion demonstrator.

The 'Bug' class defines specific features of the Gordion Bug applications. In the current versions, each of the three demonstrator applications have been separately modified. Based on experiences gained when building Gordion, we plan to integrate further features in the class structure in order to, in a more coherent fashion, expressing the levels of the architecture in the object class system. This could be done by identifying a deeper hierarchy of classes, expressing the boundaries for malleability from the $\beta$ to the $\alpha$-level. Since the programming environment contains a statically compiled programming language, we can not make instantiations from $\gamma$ and $\beta$-level to the $\alpha$ level at run-time.

Further developments on degree of local control and malleability of the C2M will be inspired by the taxonomy for tailoring forwarded by Mørch (1995), containing the following three levels:

- **Customization**: Selection among a pre-defined set of configuration options.
- **Integration**: Soft integration by means of macros, scripts or agents executing functionality in the context of the host. Hard integration by means of a component physically attached to the calling application.
- **Extension**: Tayloring by improving the implementation of the application by adding new code. In statically-compiled languages this will require recompilation, whereas interpreted languages, or interpreter extensions to statically-compiled languages will make extensions possible without recompilation.

This body of research within the object oriented paradigm could provide important input for further development when related to the architecture of the notation.

The use of high-level object-macros for implementing user interface resources as provided with the DynaWare innterface toolkit actually provides an environment for modifying the interface at the $\beta$-level at run-time, as described previously in this chapter.
8. Reflections and future developments

We have in this section collected some findings that also could be interpreted as requirements or a sort of ‘wish list’ for further development. This collection of issues is derived from the experiences when implementing the demonstrator. This section also discusses future possibilities for realizing the notation offered by state-of-the-art development of the Apple OS.

8.1 C2M techniques

**Distributed component architecture**: The construction of the demonstrator shows that it is difficult to build and maintain a single entity of an application which includes and handles all the features of a protocol. If these single entities are copied to different workstations having all the information of the protocol, the complexity of the application increases rapidly depending on the connections between these single entity applications. A distributed component architecture would be very useful for downsizing the complexities of the protocol and all the facilities required. The components would consist of parts of the coordination mechanism distributing the protocol as implemented in the demonstrator. One way to do this division is to use a role as a divider and convey the C2Ms depending on the roles. Naturally, this approach raises new questions, but apparently the protocol is then easier to implement and understand. The questions mentioned would be concerning at least updating and awareness of the changes in the protocol, how to propagate the changes to different workstations, how to be aware of the situation of some C2M in the network, how to get overall picture of all the C2M which have been done or are pending.

**Agents**: These questions lead us to considerations of the facilities the applications could have. According to the requirements we would need aggregations of the bugs to support awareness in the testing ensemble. The distributed nature of the applications does not help very much in creating this kind of information. We could have implemented a database with on-line updated information about the C2M. But some information could be handled by agents which are querying the information from the network. If required, the agent could start, e.g., mapping of the roles which are available on the network. But, what is more important, we could also have an agent monitoring the status of the distributed protocols, and thereby be aware about the C2M's status. By querying the status of each part of the protocol through linking, the agent could collect and form a dynamically updated view of the protocol. Agents could also handle the propagation of changes, thus updating of the protocol. Consequently, some basic features for this kind of communication should be built as a part of the distributed C2M. Furthermore, we could also, through agents, be aware of the organizational structure which is reflected in active applications on workstations in the network. Thus, we would be aware of new links in the network which would be interested
Architectural Issues in Design of C2M for Software Testing

August 1995

page 249

to attach some role in their workstation. The organizational structure thus could be used also as actors in nominal situation, and attached to be part of actual situation. Easy drag and drop feature would fulfill this scheme if different applications and components are to be linked and combined—this will be discussed further in Section 8.2.

**Editors:** In the demonstrator we have examples of possibilities for exercising local control. Not only with the protocol, but also to control the user interface locally. This implicates that we probably need an approach for local control of the user interface in the finalized version, like of the DynaWare-toolkit has been for us until now. Thus, an end user could change the given user interface as much as he or she wants to, hiding and showing fields, changing places of information etc. This possibility must be restricted because the scope of changes in the α-level will not allow adding functionalities. Adding buttons would require more information about the program level functionality. Because the protocol is distributed, there might be need for a protocol-editor. The functionality of a protocol editor would allow the user to edit the behaviour of the applications at β-level. A graphical presentation of the protocol, an editor which can be used also to define the rules for triggering mechanism and also connecting the information from the field of work to the user interface. This kind of editor is probably very difficult to build and the concept would need a considerable effort of design and implementation.

**Linking:** Besides checking the internal values, the Gordion demonstrator is able to receive *update* and *read* requests. In this case the demons are in the form of Apple-event handlers. As explained in the previous section, Apple-Events form the basis of the demonstrator and these high level Apple-Events can be used to interact with other artifacts. It is planned that the next versions of the demonstrator will fully support high level Apple-Events, i.e., be fully scriptable, and hence comply with the standards for interaction among Apple OS applications. A scriptable demonstrator will provide a basis for us to explore the full potential of using high level events as a means of linking computational coordination mechanisms, and for integrating the behavior of these with other applications. For example, it could be possible to define links between an Excel the software planning spreadsheet and the software bug binder in such a way that changes to one of them could invoke events updating the other.

### 8.2 Possibilities in state-of-the-art software architectures

This section briefly outlines how the design decisions made when designing the Gordion demonstrator can be seen as paving the way for utilizing state-of-the-art in software architecture. Since we have chosen to develop the Gordion demonstrator on Apple Macintosh computers, and because Apple has promoted collaboration support as one of the most important strategic initiatives, we will in this section only relate to possibilities within this frame. The main topics addressed are the Apple Open Collaboration Environment (AOCE), The Collaborative Information Suite, component software and OpenDoc.
8.4.1 Apple Open Collaboration Environment (AOCE)

Apple Open Collaboration Environment (AOCE) (Apple Computer, 1994; Buttin, 1994) is an extension to the Apple Operating System (Apple OS) providing a set of templates or catalogues defining a standard for interapplication communication. The extension is meant as a communications hardware independent standard for establishing user-defined catalogues accessible over networks of any kind. It can provide functionality similar to the yellowpages in Unix but can also contain user-defined tables and be augmented by code resources. By combining AOCE’s mailing and catalog functions it is possible to route documents from any Macintosh application. If the receiver does not have the application on which the document has been created, it can be read using a standard document reader (Apple Computer, 1994). AOCE contains a Standard Mail Package, a Standard Catalog Package and the AOCE Toolbox which is a Digital Signature Manager, an Authentication Manager, a Catalog Manager and an Interprogram Messaging Manager.

The AOCE operating systems extension could provide a framework for implementing organizational directories, such as human resources in the Foss Electric case, which could be accessible over the network. These directories could be linked to, for example, the bug forms in Gordion which could be routed by actors simply by drag-and-drop. The authentication feature can support the definition of roles and implementation of protocols. In general, AOCE templates could provide a foundation for implementing the objects of articulation work (OAW) defined in the notation. The OAW functional primitives could be expressed by added code resources to the templates.

The advantages of building on top of the AOCE framework is, firstly, that it is a standard promoted by Apple—one of the most advanced operating systems developers, secondly, that the C2M’s complying with this framework will function independently of means of communications, be it ethernet, modem connections or microwave connections of Personal Digital Assistants (Apple Computer, 1994). The architecture of the Gordion demonstrator as role-defined active artifacts where forms are stored as documents is consonant with the architecture of AOCE templates in the sense, that a bug-form can be made part of a workflow system defined in AOCE.
8.4.2 The Collaborative Information Suite

The Collaborative Information Suite (Apple Computer, 1993) is not a piece of software. It is an object class hierarchy defining a standard for:

“Apple event constructs for applications which deal with contacts, resources, locations, and addresses … it can be used in conjunction with other suites (such as ones which define constructs for time, events, projects, etc.) … This suite is also designed so that it can be used in conjunction with the Telephony and Mail Suites to support the integration of various applications in personal communications, workflow and other collaborative solutions” (Apple Computer, 1993).

The Collaborative Information Suite is work in progress and the document describing it is a preliminary developer note. The purpose of the object class structure is to define a standard for data resources in applications which supports collaboration. The rationale being that if developers of these applications all comply with the standard, then the various Macintosh applications produced will implement a standard for interapplication communication ensuring cohesion. Several scenarios describing use situations are outlined, such as:

“A magazine editor could type some buzzwords for a story into an outliner. A contact manager could use these to search through a “notes” column and return a list of all the people that match the buzzwords, creating a source list that can be sent to a reporter.” (Apple Computer, 1993)

The suite defines object classes for records containing information supporting collaboration, i.e., by complying to the standard applications will contain information from both the cooperative work arrangement and the field of work. It also defines entities for addresses. The Collaborative Information Suite also defines AppleEvent protocols for accessing the structures. The developers note describing the Collaborative Information Suite lists applications that directly address AOCE catalog services, applications that manage contact information and applications that manage resource information as applications which should comply to the standard. As examples of client applications, the note lists: Calendar and scheduling applications linked to a contact manager, word processors automating mail merge and simplifying address retrieval, and various information managers such as spreadsheets and databases where the information in a contact manager will not be duplicated. A scheduling system linked to a contact manager can easily be viewed as two linked coordination mechanisms, reflecting a granularity of mechanisms quite similar to the one shown in the diagram in Figure 13 where the human resource coordination mechanism is viewed as a contact manager.

A combination of AOCE and the Collaborative Information Suite can be seen as providing a relatively high-level computational foundation for building organizational handbooks and directories supporting indexing of the organizational context of cooperative work. In this respect the suite could form a basis for building coordination mechanisms containing information on various kinds of resources: human, technical, informational, etc. The object class structure resembles the objects of articulation work (OAW) defined in Strand 3 of COMIC, but does not contain all elements and some elements are defined differently.
Furthermore, one of the critical points which can be raised when comparing the suite to the notation defined in this deliverable, is that the Collaborative Information Suite defines entities at the semantic level of implementing systems which contain information about persons, addresses, phone numbers, locations, organizations, etc. An obvious future task will be to take a closer look at the Collaborative Information Suite, analyze the differences and attempt to define the OAW in terms of the Collaborative Information Suite if possible.

8.4.3 OpenDoc

Developments of standards, tools and applications within the paradigm of component software or interoperable objects can be seen as attempts to end the era dominated by a relatively few monolithic applications. Since the first IBM PC or Apple Macintosh more than 10 years ago, the programs end-users run on their computers in order to do word processing, spreadsheet calculations, etc. have been feeding higher demands for RAM, harddisk space and computational power with a rapid growth in built-in features. Word processors contain elements of drawing programs. Applications each implement spell-checkers. Spreadsheet programs contain functionality for building interfaces, and also contain simple database functionality. Applications such as Ragtime and Lotus 1-2-3 were collections of applications bundled in order to ensure easy transfer of data from one application to the other. The first Apple Lisa and Macintosh computers defined standards for easy cut-and-paste data exchange between applications—this was a major step forward. Later on Microsoft Office and WordPerfect Office, because they each were developed by one company, provided functionality for integrating elements from one application to another, such as embedding a dynamically updated spreadsheet element in a word processing document. Efforts like these solved some of the problems users encountered when they needed to mesh work products from several applications. Several problems were, however, not solved by these integrated applications. For a person working with a computer, the complex of applications each demanding several megabyte RAM is not necessarily the most natural way of conceptualizing work. If you, for example, are producing a magazine, the most important entity is the magazine containing text, pictures and illustrations, not the applications representing modern computer variants of typewriters, light-tables, drawing boards, rulers and repro-equipment.

To make a very long story short, several companies and conglomerates are working on providing the technology placing documents in focus. Ole from Microsoft is one such development, OpenDoc from Component Integration Laboratories is another (Valdés, 1994). These developments can be viewed as an attempt to place the work artifacts in the center and, in the words of Wirth (1995), to provide leaner software, where the user can configure the support needed.

The basic idea is to get away from large complex applications which cater for every wish from every market segment. OpenDoc is an enabling technology which provides the first important step in a different direction:
OpenDoc restructures application development in a way that fosters small, reusable, interoperable application components that users can mix and match according to their precise needs. OpenDoc is an open architecture for the creation of compound documents—which can contain many different types of data (such as text, graphics, tables, video, and animation). The documents can be edited, printed, circulated in read-only form, presented as a slide show, and circulated for mark-up and review. Users can customize their work environment by discarding, adding, or replacing objects with those from other vendors. (Rush, 1994)

The contents in OpenDoc documents can be dynamically and automatically linked to databases. Scripting functionality is provided for allowing users to collaborate:

These scripts do not have to be low-level macros that only contain raw information such as mouse clicks and keystrokes. Instead, the scripts can encode operations that reference the parts of a document semantically—by section, paragraph, word and so on—Independent of which application object is responsible for which content type. (Rush, 1994)

Interoperable component environments can support designers and end-users in building and tailoring systems by supporting management of dependencies among different work products (Sumner & Stolze, 1995).

Future work on the Gordion demonstrator will include testing the possibilities of expressing C2M functionality as OpenDoc components, thus integrating computational coordination mechanisms into OpenDoc documents supporting cooperative work, and to provide linking between mechanisms by defining them as OpenDoc components. The architecture of the demonstrator is prepared for conversion into OpenDoc components for three reasons: 1) the demonstrator consists of a set of applications with limited functionality, 2) the document is the basic work product entity, and 3) the basic architecture is defined as an object class structure.

The Project Schedule used at Foss Electric is an Excel spreadsheet. An OpenDoc spreadsheet component could be linked to C2M components supporting the coordination of the software testing process in a seamless and tailorable manner. By embedding C2M functionality into interoperable components, the demonstrator development will be able to directly reuse components developed by others—as noted by Brooks (1987), reuse of code is one of the viable ways of obtaining improvements in software productivity.

The idea of viewing computer support as a matter of combining pre-defined components in relation to documents is very appealing in the context of the rationales behind the work in Strand 3.

Acknowledgments

Thanks to Thomas Albert, Liam Bannon, Peter Carstensen, Kjeld Schmidt and Carla Simone for valuable discussions.
References


Chapter 8

Demonstrating a cooperative system with an extended single user GUI

Tuomo Tuikka

Department of Information Processing Science, University of Oulu, Finland
ttu@rieska.oulu.fi

During spring and summer 1994 a HyperCard based prototype software application was developed at University of Oulu for creating demonstrations of cooperative systems. This MOI-tool enables users to link user interface objects like fields, buttons, cards etc. between many workstations by slightly extending HyperCards facilities and its graphical user interface (GUI). The objective of the MOI-tool was to support demonstrating cooperative systems in a participatory design situation and the requirements to build it were derived from a small setting of case examples, whereas the ideas for implementation are principally loyal to user interface solutions of Hypercard. This chapter discusses of experimentally demonstrating a cooperative system: a bug handling mechanism, in order to discover differences between the MOI-tool design and the requirements of a possible cooperative system. The experiment results in one exemplary demonstration of a cooperative system with an explanation of breakdowns during this demonstration. These problems illustrate apparently the shortcomings of using user interface objects as a basis for the MOI-tool design for demonstrating cooperative systems. The paper ends up in a consideration of a possible future approach which could mean a change in concepts of tool design to another level of abstraction, in the level where we are discussing of roles, and actors, the objects of articulation work.

1. Introduction

MOI-tool is an evolving prototype fulfilling some ideas of how a user can build a demonstration of cooperative user interface. Basically it is now a HyperCard based application in the sense that it is mainly implemented by using HyperTalk, but on the other hand it exploits advanced features of networking by allowing linking of user interface objects between two or more workstations.

Our first design objectives served two purposes, first the tool was to be considered as a fast prototyping environment for creating demonstrations of cooperative systems in a participatory design situation. Furthermore, a possible use could be supporting users to perceive the organizational context of collaborative work and corresponding information processes (Hellman, 1990). Second, the research purpose was to experiment with a set of primitives satisfying the basic operations for creating a demonstration and finally to study the
relationship of these experimental primitives with the notations, describing mechanisms of interaction used in cooperative work.

However, the motivation for this paper is due to the design that the tool has been built on. We would like have a tool to demonstrate cooperative systems, but the problem and apparent weakness is that principal design ideas for MOI-tool are derived from single user interface environment and only four exemplary cases of cooperative work situations. Thus, another approach has been chosen from the field of Computer-Supported Cooperative Work (CSCW) to inform this design and later on possibly in the form of requirements to be considered in further evolution of the MOI-tool. In this sense the MOI-tool is not only an environment for fast demonstration platform for cooperative systems in participatory design situation, but during this evolution also possibly a source of new ideas of e.g. user interface design for cooperative systems.

In CSCW a crucial issue is to devise computational coordination facilities that provide support for cooperating actors in managing the complexity of articulating their distributed and yet interdependent activities. 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) Because multiple actors are involved, cooperative work is inexorably and fundamentally distributed, 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, perspectives, conceptualizations, goals, motives, etc. (Schmidt, 1991).

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. As a generalization, these artifacts and the concomitant procedures and conventions are called ‘mechanisms of interaction’. A mechanism of interaction (MoI) 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 (C-MoI) can be 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, 1994).
In a field study of software testing (Borstrøm et al., 1994; Carstensen and Sørensen, 1994) present a bug handling mechanism of interaction which has been selected as an example to form the basis of this analysis. Through an experiment we will take a look at how MOI-tool can be used to demonstrate the bug handling mechanism in order to find out any particular concepts necessary for demonstrating this kind of cooperative system. As a result there will be a presentation of one possible way to demonstrate a bug handling mechanism with a discussion what could be necessary concerns in thinking of requirements for the tool. To avoid confusion it is essential to notice that the aim is not to build a mechanism of interaction or an elaborated design of one but only a demonstrator, as well as the aim is not to suggest a design of the MOI-tool which would answer to the problems encountered during the experiment. However, this paper argues that there is a gap between the design of this tool and the concepts of bug handling mechanism which emerges from the reason that we use a single user interface concepts to present a demonstration of a computer supported cooperative bug handling mechanism. This argument in my opinion has implications to the need of reconsiderations in the way we think about user interfaces of cooperative systems. That discussion is out of the scope of this paper, though. Furthermore, from now on the computer based bug handling mechanism will be referred to as the “bug MoI”. It is also presumed that the reader has basic knowledge of HyperCard™, a trademark of Apple Computer, Inc.

2. Research approach

During the design and construction of the MOI-tool we had an idea of how the tool should work in four case settings of exemplary cooperative systems (Tuikka et al., 1994). These examples were first transformed to simple drawn scenarios illustrating the user interface in terms of user interface objects familiar from HyperCard like stack, card, button, field, background etc. Next step was to think about linking these particular objects. The assumption was at least to link them so that the same information could be seen in fields of other machines.

Finally, a primitive language was developed to serve the purpose of linking and also to provide additional functionality to the demonstration. These were added for reasons like needs to change a card to another card from other machine or to launch sequence of action from objects in other stacks for example. This approach seemed to be enough for the examples we had when we added a new object, scrolling list, to the user interface. With this new object type and its functionality it was possible to demonstrate timetables or even a small shared spreadsheet.

But would this approach be usable more generally and not only in those situations we had ourselves created. This suspicion can be presented in one sentence: there is a gap between the design of MOI-tool and the approaches for requirements and design for systems of cooperative work. If this is true in one
case does not prove it to be true in all possible cases but should certainly produce at least one good argument on what MOI-tool approach can’t be used.

Support for articulation work is shared by many researchers in the CSCW community like e.g. THE COORDINATOR (Winograd and Flores, 1986; Flores et al., 1988), DOMINO (Kreifelts et al., 1991), EGRET (Johnson, 1992), CONVERSATIONBUILDER (Kaplan et al., 1992), OVAL (Malone et al., 1992). While closely related, the approach of (Schmidt, 1994) 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. In this paper we are interested in the field study which was made to serve establishing requirements for a specific computer based mechanism of interaction supporting a specific work setting. In his work (Carstensen, 1994) argues:

“Designing computer systems supporting articulation activities requires a deep and coherent understanding of the work domain to be supported.”

In the same manner it can be argued that designing a tool for demonstrating distinct articulation activities it is required to understand not only the work domain but also articulation activities in question. Thus, the field study has been selected as an input for the analysis.

All in all, a research question is that how does the design of MOI-tool differ from the needs of the bug-MoI, in order to demonstrate it as a computer supported cooperative system? Thus, it is necessary to discuss first about the MOI-tool and second the requirements of the bug-MoI. Furthermore, an experiment is used to demonstrate how is it possible to present the concepts of the bug-MoI. Finally, there will be a discussion about the findings of the study.

3. MOI-tool basics

MOI-tool is at simplest a HyperCard stack, but its additional features give it a strength of easy and fast creation of demonstrations of cooperative systems in several networked workstations at the same time. As additional features to normal HyperCard stack, MOI-tool allows user to build objects that can be linked between workstations. Furthermore, primitive functions can be used to add more functionality to objects during the linking process. A symbiosis with HyperCard allows a platform for rapid tool construction and modification as well as a supporting platform for creating a demonstration.

The design was derived from the requirements of MOI-tool that we agreed in order to build a tool for a participatory design situation. One of the basic requirements is that the user interface for the MOI-tool should be easy to use. This means that we wanted to be consistent with the environment selected, since the users were supposed to be familiar with HyperCard. Furthermore, there was an idea that it should be possible to connect objects in different screens to each other,
which requires us to be loyal to the user interface solutions of this environment. Thus, the MOI-tool was built to slightly extend the concepts of HyperCard and familiar objects like: card, field, button etc. were used. But how to add functionality to the objects? We added a feature which we call primitives which can be attached to user interface objects. A similar research approach can be found from (Dewan and Choudhary, 1991) where primitives were created in their experimental work building multi-user applications as an extension single-user applications. However, the important difference here is that the MOI-tool addresses and uses the user interface objects due to the requirements instead of users or groups.

Finally, we experimented with this primitive language and added an additional object, scrolling object, for easier creation of cooperative tables etc. Next chapter will explain a little further how these familiar objects are made cooperative and illustrate what is the difference between HyperCard and MOI-tool.

3.1 Objects and their management in MOI-tool

The HyperCard user interface has been originally developed for a single user and it is famous of being easy and useful for building applications, some of which are even advanced database applications. For further introduction of HyperCard environment it is recommended that you read HyperCard help files (Domurat, 1991).

In order to find out how do we use the HyperCard concepts we must take a view to them considering what has been added and how. For instance, HyperCard fields are objects that contain information in either textual or numerical form. With MOI-tool, by attaching and combining primitives, it is possible to add new features to a particular field. A combination of these primitives perform an action. Action could consist of a single or several primitives and they can be triggered in many different ways, simplest of the triggers is the click of mouse button. Thus, buttons can be used to start actions that affect other interface objects. These actions can be launched also by addressing a specific launch-primitive to the object in question. The difference regarding normal user interface objects is that we can link these primitives not only in one workstation, but anywhere in the network.

If these new features are desired to be used, the activities to handle or manage objects must be accomplished using MOI-tool menu. This is the only visible part of the tool unless you choose dialogs from the menu. These activities are thought to be of general nature considering graphical user interfaces and they are also needed for user control of object handling. We can indicate following activities: Create, Define, Change, Destroy. Some of those activities can be done with HyperCard and some with our additional tools included in MOI-tool as you see from following explanation.

Create: A “cooperative” object must be created by choosing Toolbox-menu from menubar and choosing create field etc. from a pull-down menu. Although
objects look like the same when created from normal objects-menu, they possess additional properties. **Define:** If cooperative features are desired the properties of object must be defined by using ObjectTool which also is our addition to HyperCard. Both primitives and their links can be attached and defined by ObjectTool. ObjectTool can be used by choosing Toolbox-menu and ObjectTool from menubar. 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, changes to cooperative features after use of ObjectTool are possible only from HyperCard's script editor. **Destroy:** Objects are deleted with HyperCard's own tools.

3.2 Primitives of the MOI-tool

In our terminology a set of primitives means such a set of primitive functions implemented inside the MOI-tool stack that can be used to build up all our described and required functions in the cases we had as a requirement, thus implying no generality. The choice of such primitives was as an approach iterative. To make primitives simple, they are built in such a way that there are no connections to other primitives, i.e. they are self-sufficient. This has been a problematic requirement during the work because sometimes information about the state of environment, what the other primitives have done might be needed as an input (Tuikka et al., 1994).

After a user interface object has been 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, by for example pressing <Enter> while cursor is in a field.

![Figure 1. Primitive of user interface object clears field in other machines user interface](image)

To get an idea of how linked object works take a look at figure 1. In this example object is triggered by e.g. pressing mouse, the script of the object starts running and primitives which has been attached are run. Attachment shown includes a clear-primitive which addresses other user interface field for clearance. Another widely used primitive is “send” for sending contents of a field to another
field. It is notable that the primitives use terms such as send and the commands are connected to user interface objects as seen in the figure 1. Other implemented primitives which can be pointed at the field object are: Clear, Date, Time, Launch, Lock, Send, hideObject and showObject. This list is only exemplary for field object and we have defined 28 primitives which are depended on the object where they connected (Tuikka et al., 1994).

4. The Bug Mol

It is essential in software testing work to document and to be aware of errors in a software product. Thus, it is usual to use a bug handling form in order to register an error and inform other people of the errors. This kind of bug handling mechanism has been presented in (Carstensen and Sørensen, 1994). The study is interesting and useful for us because a set of requirements and a detailed presentation how this mechanism of interaction should work is available. Although the example is from a distinct study, it has been changed to be suitable for this work and all the changes are in the responsibility of the author.

In search of what we are going to demonstrate we need first some ideas how the handling of software bugs can be performed. 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. Several groups of actors are involved in this process. They are thus users of the form and the procedures.

The information flow described next illustrates the intended flow of the bug handling mechanism. The flow (the route of the forms) follows seven major steps: (1) A tester sends a form describing a recognized bug to the specification team. (2) The spec-team adds diagnosis and estimation information and sends it to a software designer. (3) A copy is sent to the central file manager. If a bug is rejected the original is sent to the central file manager. (4) The software designers add correction information to the form and send 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 sent to the central file manager. A design of one possible computerized bug handling mechanism user interface is presented in Figure 2.
The design of the demonstration follows this layout carefully and can be seen later in this paper. There is no attempt of redesign or suggest any improvements, but only use it as an example of a user interface while the numbering has been used as a reference. The set of requirements for the computerized bug handling mechanism of interaction implies us what issues we need to address in a demonstration.

Requirement 1: 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.

Requirement 2: 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.

Requirement 3: 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. 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.
**Requirement 4**: 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 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.

5. The Bug-MoI demonstration

In the demonstration the MOI-tool is used to illustrate how the cooperative work happens in a software testing environment with registering-routing-diagnosing-correcting and verifying process, viewing the process of handling bugs from the actors perspective. Thus we can conclude later on does the demonstration provide the facilities or services required as requirements.

The presentation shows how certain kind of demonstration would look like and a detailed example of interaction in step by step fashion between the actors and the computer based bug handling mechanism. This is also the basis of the experiment; a description of a 'typical flow' of the registration and correction process. Each row in the tables 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. Let us now see what kind of demonstration we will have. The artifact must be drawn and objects attached.

<table>
<thead>
<tr>
<th>Actions from the actor(s)</th>
<th>The responds from the Bug-MoI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) A tester recognizes a bug in the software, decides to report it, and “generate a new bug report”.</td>
<td>The Bug-MoI replies by setting up an electronic form containing entry fields for the relevant information.</td>
</tr>
<tr>
<td>2) The tester classifies the bug, fills in the fields in the form, and ask the system to “register the bug”:</td>
<td>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, it is filed in the central bug database and a notification is send to the spec-team.</td>
</tr>
</tbody>
</table>

Table 1: A table of the typical tester actions and bug-MoI reactions (each pair in the rows) in the registering-routing-diagnosing-correcting-verifying process.

Demonstrating the first entry is rather easy to fulfil. A platform is quite easily drawn and entry fields can be built and filled beforehand, which means that the default values are easily filled in to show that the bug-MoI has the information or is linked to another mechanism. Report number (field 4 in figure 2) should be automatic and not editable. Initials depend automatically on who has logged in. Instrument has a default value, and this field is thought to be connected to a repository of pending projects. Date should have a default value (field 2).
In entry 2 fields error type (5a) and importance (6a) are filled by tester, and the data of the bug is saved to a database. The bug-MoI in this case checks all the fields which are filled, if they are not a request would pop up. The demonstration does not have any database and all the data is saved in the fields. Checking the fields is not easily done, one has to do HyperTalk scripting for field checks. Apparently a field checking feature would be helpful for the MOI-tool. The command “register the bug” should be illustrated somehow as well.

Then the actor changes. A notification should go to the spec-team and the spec-team member asks for the next new registration. After steps 1 and 2 a new registration is ready to be shown in the other workstation, which is considered as the spec-team member workstation. Notification could be presented anyway, but in the demonstration the presentator must explain that now the change has been notified and spec-team is receiving the registration information (entry 3). We can express the status of the bug with indicator change in item 22 to 24 (see figure 2).

The demonstration bug-MoI has a button attached to the demonstration functionality field (cf. figure 3) that has the ability to clear all necessary fields and another one for launching the transmission of data from tester's user interface.

<table>
<thead>
<tr>
<th>Actions from the actor(s)</th>
<th>The responds from the Bug-MoI</th>
</tr>
</thead>
<tbody>
<tr>
<td>3) A spec-team member asks for the “next new registration”.</td>
<td>All registration information are presented to the spec-team member together with relevant fields to be filled in concerning the diagnosis and estimation.</td>
</tr>
<tr>
<td>4) If the bug cannot be accepted by the spec-team, a spec-team member demands to “reject bug”.</td>
<td>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.</td>
</tr>
<tr>
<td>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”.</td>
<td>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.</td>
</tr>
<tr>
<td>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).</td>
<td>On the basis of the specified module, a default responsible designer is added to the information.</td>
</tr>
<tr>
<td>7) The responsible designer(s) are filled in together with the correction time estimate for each. “Bug accepted” is demanded.</td>
<td>The bug is filed as “accepted” in the central database. The responsible designer(s) and the originator (the tester) are notified.</td>
</tr>
</tbody>
</table>

Table 2: A table of the typical spec-team actions.

Entry 4 in the list is a rejection of the bug, and a central database should be updated. The demonstration has an indicator for rejection (item 16 in figure 2) which can be selected for showing this status change, however saving to a database must again be explained by the presentator since no database exists.

HyperCard has a feature for grouping user interface buttons, but unfortunately it is not possible to connect the grouping to another machine. MOI-tool doesn't do
that either, buttons and their actions cannot be connected together to show selections. This could be one possible user interface feature. The progress of entry 5 is like in the previous entry. The notification could be of any kind, a message or a status list that can be opened anytime.

In case of acceptance, fields (entries 6,7) 6b and 5b are filled, item 16 in the user interface (figure 2) is set to accepted and some default values appear to fields, platform period, when the bug must be fixed and the responsible modules are set. Depending on module a responsible designer could be set to modules field (item 7 in figure 2), thus demonstrating linking to some other mechanism of interaction would be available.

When designer wants to see correction requests (entry 8, table 3), the bug is shown and presented with status “correction”. Again, in the demonstration when the action is triggered by the user, all the information is sent field by field to the other workstation, which means that routing is all the time only demonstrated. In entry 9 the designer can reject to do the correction. Entry 10 means that the designer accepts the diagnosis and a notification goes to database and spec-team members. The demonstration doesn't show that. However, real mechanism of interaction would result in routing these notifications to the required people according to the information of the roles and actors.

<table>
<thead>
<tr>
<th>Actions from the actor(s)</th>
<th>The responds from the Bug-MoI</th>
</tr>
</thead>
<tbody>
<tr>
<td>8) A designer demands a “see correction request”.</td>
<td>All information on the registered bug is presented to the designer.</td>
</tr>
<tr>
<td>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”.</td>
<td>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).</td>
</tr>
<tr>
<td>10) If the designer accepts the diagnosis and estimate he demands a “accept request”.</td>
<td>The bug is filed as “correction request accepted” in central database and the spec-team members are notified.</td>
</tr>
<tr>
<td>11) The designers asks for a “register correction”.</td>
<td>Identification information on the bug is presented to the designer together with fields for registering information on the corrections.</td>
</tr>
<tr>
<td>12) The designer fills in the time spend and information on affected modules and files, and demands a “register correction”.</td>
<td>The bug is filed as “corrected” in the central database.</td>
</tr>
</tbody>
</table>

Table 3: A table of the typical designer actions.

In entry 11 the designer registers a correction to the fields (item 19 and 20-21) and in entry 12 all necessary information is filled and saved to the database. In the demonstration presentator can fill the fields and explain what happens. The status changes to verification and a new actor, platform master, takes over the situation. In entry 13 an aggregation of corrections is presented to the platform master which is not possible to present by the demonstration and in entry 14 the demonstration shows the information of the next bug to the platform master. All
the other possible actions of this actor can be explained in the same way as in previous situations.

<table>
<thead>
<tr>
<th>Actions from the actor(s)</th>
<th>The responds from the Bug-MoI</th>
</tr>
</thead>
<tbody>
<tr>
<td>13) The platform master asks to “see corrections to be verified”.</td>
<td>Information on all corrections to be verified in the next platform integration period is presented to the platform master.</td>
</tr>
<tr>
<td>14) The platform master demands a “register verifications”.</td>
<td>Information on the next bug to be verified is presented to the platform master.</td>
</tr>
</tbody>
</table>

Table 4: A table of the platform master actions

The demonstration version of the “computerized” bug handling mechanism artifact is seen in figure 3.

![DemoWindow](image)

Figure 3: Designers user interface created with MOI-tool. Special controls for demonstration have been added below in the picture. They are a function to: get a new correction bug, send all the information to the platform master and clear fields for next demonstration.

The layout follows the design presented earlier and shows the MoI as seen by the designer. Only the lowest part of the picture has new additions, which belong to the demonstration, since the demonstration situation must be controlled somehow.

6. Discussion

It is not possible to say whether the recent design of the MOI-tool would be good or bad. Although we gain in using the HyperCard user interface objects for
enhancing it to multiple workstation use, it is not very flexible in demonstrating the bug-MoI and its routing process. It seems like we are mostly using the features of HyperCard for demonstration, and MOI-tool for presenting routing while the demonstration rigidly stays in two workstation. Also we must closely follow the demonstration “manuscript”. We benefit on the feature that the layout can be drawn with the HyperCard tools which is easy and standard way of building an interface in HyperCard. Normal HyperCard environment in the form of fields and buttons can also be used for presenting features of a single workstation.

The idea of the demonstration was to study what kind of difference we can find from demonstrating a cooperative system, a computerized bug handling mechanism with the design used in our tool. These situations where difference occurs could also be called as “breakdowns”, where the tool cannot fulfill the wishes. The requirements presented earlier can be seen as a demand for ultimate bug-MoI and are now helping us to compare what we have gained from the example.

**Requirement 1:** The demonstration does allow registering of a new bug, and it is possible to demonstrate distributed registration, although restrictedly. We can at least show what happens when an actor registers a bug in one of the roles. But nothing checks mandatory information fields. This feature can be seen already in entry 2 of the demonstration. Filing and aggregations are not possible since there is no database feature.

**Requirement 2:** The demonstration is able to present the routing of the information between the actors. However, the actual structure of the demonstration is so different that it only sends the information field by field, no checking is available and information is not automatically passed. Therefore, routing must be triggered from a button, cf. figure 3 button: send all to platform master or e.g. entry 3. The information that has been filled and selected are then sent to the other user interface, which in turn must be set so that the primitives know where to link the sent data. The information is not registered, but it is passed on. A notification is missing from the demonstration, but it actually could be realized. However, the notification rejection would not be routed according to information of actor or a role, but crudely to another workstation.

According to requirements the actor completing an activity must be able to overrule the routing and redirect the information to whoever he or she wants. An actorlist or rolelist could be presenting this possibility.

**Requirement 3:** Demonstrating resource allocation is limited but can be explained and shown as an example, the same applies to the software complex etc.

**Requirement 4:** Aggregations could be demonstrated as drawings and written presentations without any special features.

As we can see there are many points showing breakdowns in the demonstration. Checking mandatory field information could be possible in the sense MOI-tool handles information. However, consider the routing of the bug-MoI especially, a function where we used MOI-tool extension most. The only
way to build routing is to use clear and send-primitives and change cards in different machine. This leads us to a wide use of same commands since there are many objects to link. In one screen of 20 fields we need 20 clear-primitives and the same amount of send-primitives to update a demonstration screen. Only to work with that number of links is hard work. This is a problem we didn’t think of.

Finding out that finally we are using only three of the 28 primitives leads into suspicion. Apparently the primitives work in different level of abstraction, when we connect user interface objects instead of discussing actors, roles, i.e. the objects of the articulation work. This seems to be relevant throughout the example, it is possible to show resource allocation discussing in terms of roles and actors and can be shown as explained before, but a demonstration must be built in the same way than the other user interfaces. No additional help would be available from MOI-tool. This is one of the most difficult problems and thus it probably leads us to reconsiderations about in what level we should discuss in building a cooperative user interface.

All in all there seems to be at least some ideas for improving the MOI-tool. Like adding a field checking feature with some kind of condition attributes or implementing a feature for facilitate notification to somebody. Also grouped controls, not only in one workstation but between the workstations could be helpful and possibly somekind of database feature. Finally, a feature for helping copying the cards from workstation to another easily and copying the MOI-tool to another stack using AppleScripting would be helpful.

Conclusion

First and foremost, we have found breakdowns difficult to work around, therefore we have a good evidence that MOI-tool is not a general tool in representing CSCW systems. However, it can fulfil several different requirements that make it do — what it at best can do — serve as supportive platform when building a demonstration and help with discussing about the properties of a cooperative system.

But is something else needed? Consideration of new requirements is necessary since the example used the features of MOI-tool scarcely, but are there new useful concepts in user interface design behind the corner? Like utilizing semantic level of the articulation work as a useful concept to move from user interface objects to overcome this complexity seen with user interface objects.

Acknowledgments

The author wishes to thank Carsten Sørensen and Peter Carstensen for discussions about the bug form mechanism, F. C. T. D. (Tekes) in Finland funding this work in the Finnish part of the ESPRIT project 6225 COMIC and IRIS 18 reviewers for comments.
References


Chapter 9

Design and Use of Mechanisms of Interaction

B. J. Wielinga, R. de Hoog, P. van de Besselaar, J. Wielemaker
University of Amsterdam

R. Gustavsson
SICS

1. Introduction

This paper describes methods and tools for the development of CSCW applications. The development method is based on a framework for analysis of cooperative work that was used to analyse a number of case studies in Social Mechanisms of Interaction. On the basis of this framework and the notion of Mechanism of Interaction (Schmidt:94a) a software tool was designed and implemented that allows rapid construction of CSCW applications. Before we discuss the framework in section 2, a number of general issues concerning the notions of cooperation and articulation are discussed. The tool is described in section 3.

Cooperation and Articulation

In this section we discuss the notions of cooperation, articulation and mechanisms of interaction as described in Comic Deliverable 3.1, partly to enhance our own understanding of these important concepts, partly to contribute to further developments of the ideas.

In Schmidt(93a) cooperative work is defined as work for which ``multiple actors are required to do the work and therefore are mutually dependent in their work and must coordinate and integrate their individual activities to get the work done.’’ The notion of interdependence is the crucial characteristic of cooperative work. Three qualifications should be made with respect to this definition. First, the definition implicitly assumes a bounded system that is the object of study. In the large, all of mankind is cooperating in keeping this world go round. In the small, each person is independently acting according to his or her goals. Neither of these extreme views makes much sense in the context of CSCW. Explicating these boundaries makes clear what can be considered cooperation and what cannot.

A second point concerns the notion of interdependency. Actors can be interdependent in a narrow and in a broad sense. Narrow interdependence occurs
when actors need products of the activities of other agents in order to successfully complete their task. Without cooperation the actors would not be able to achieve the common goal. When actors are not directly using each other’s results, but share or compete for resources one could speak of broad interdependence. Drivers approaching a junction are interdependent since they share the road as a resource. Deliverable D3.1 appears to adopt the narrow interpretation of the notion of interdependence, but there does not seem to be a consensus in the literature. We will encounter examples of both forms of interdependence in later sections.

A third source of ambiguity in the notion of cooperation stems from the interaction between cooperation and formal regulatory mechanisms. If cooperation is controlled through a strong set of rules or laws (protocols), the cooperative aspect reduces to agreed upon coordination of activities and the need for articulation of work disappears altogether. For example, drivers approaching a junction in a country where no traffic regulations exists, need to cooperate in order to prevent collisions. They may adjust their speed in relation to that of the other car, or may signal to give the other driver the right of way. When a junction is equipped with traffic lights and drivers adhere to the rules, each driver can decide what action to take just on the basis of the colour of the traffic light and can ignore other cars. The strong regulatory power of the traffic light has removed the need for explicit negotiation. In conclusion: the notion of cooperation is a relative one.

Articulation, according to Schmidt(93a), is the process of dividing, allocating, coordinating, meshing, interrelating etc, of a set of activities. Articulation determines who is doing what, when, where and how. Articulation is achieved through devices of interaction: mechanisms of interaction. Below we attempt to clarify the concept of articulation and its role in cooperation by distinguishing three distinct spaces reminiscent, but not identical to the three notations proposed in Deliverable 3.1.

In the following and in sections 2 and 3 we will sometimes use a departmental meeting as an example of cooperative work involving knowledge exchange and knowledgable decision making. Meetings between members of some organisational body, such as a departmental board or a project steering group, are typical organisational devices to counter the problem of lack of confidence. Important decisions are not taken by one individual but through a formal meeting. In many cases meetings, e.g. the departmental meeting of a University department, also serve the purpose of solving the {lack of knowledge} problem. The meeting is a solution of the shared responsibility problem type. Participants share responsibility for the decisions that the meeting takes. The type of meeting that we have in mind in the example is that of a formal meeting with a chairperson, voting procedures, formal minutes etc. We will use the meeting example to illustrate the notion of different spaces described below.
Work space

The work space is the world were the actual cooperative work is performed. The work processes and the objects that these processes operate on, are located in this space. In the example of a departmental meeting, this space contains the processes of expressing opinions, submitting proposals for decisions, decision making etc. Communication in the work space takes place in the work space language. This corresponds to what is defined as the alpha notation in (Schmidt:93a). In our meeting example, the work space language is normal natural language, with some phrases specifically used for meeting work, such as ``to second a proposal',', ``veto a decision'', ``ammend a proposal'' etc.

Articulation space

The articulation space is concerned with the processes and messages that embody the articulation of the cooperative work in the work space. In the meeting example, articulation occurs when an agent requests the floor to make a statement, when the chair assigns the floor to an agent, or when the chair indicates that a topic has been sufficiently discussed and that the time has come to make a decision. Typical articulation signs and symbols are: pointing to a person to indicate his or her turn to speak, raising a hand to request the floor, interrupting a speaker to indicate that time is up etc. There are several ways in which the elements in the articulation space interact with the work space. The issue here concerns where the articulation control is.

First, the articulation signs and symbols can be embedded in the work space communication, articulation control is in the work space. Starting to speak without an explicit assignment of the floor is an example of embedding a request in a work space activity, introducing a topic is part of the cooperative work, but may also serve the purpose of indicating that a new agenda item is the focus of the discussion. In other cases the articulation control resides in the articulation space or we can have a flexible locus of articulation control.

A second interaction between articulation space and work space has a controlling function: processes in work space are started, stopped, topic boundaries are established etc.

Meta-articulation space

The signs and symbols used in articulation and their effects, both in work space and in articulation space itself are often determined by conventions or explicit stipulations, such as statutes of operating procedures (Schmidt:93a). Such conventions and rules are not part of the articulation space itself, but are located in meta-articulation space. The (meta-) model of articulation prescribes the ``rules of the game'': how should the articulation signs and symbols be interpreted in terms of cooperative actions. The semantics of cooperation can be achieved through common communication languages and agreed upon ontologies. The model of articulation can arise through repetitive ad-hoc articulations that become
conventions, through design (e.g. in the case of law or statutes) or through dynamic processes that establish or change the articulation space during the cooperative work. In meetings, for example, the meta-articulation space is entered through a `point of order’ message, after which the discussion is about the rules how to conduct the meeting.

Figure 1: Three distinct spaces in cooperative work

Figure 1 schematically presents the three spaces that were described above.

It is important to note that the distiction between work space and articulation space is a relative one. Articulation can be viewed as work in itself and when this is done cooperatively, it is itself subject to articulation. Let us illustrate this by an example. Consider the construction of a dictionary by a group of people. Each of the members of the group will produce a number of entries for the dictionary: the actual work of dictionary building. Before the work can take place a number of coordination decisions have to be articulated. The corpus of words to be entered in the dictionary has to be divided up between the co-workers, conventions about the format of entries have to be agreed upon, rules about order of entries have to be established. From the point of view of entry writing, these processes are located in articulation space, but one could also view establishing the format and conventions as `real work’. In fact, it usually is the work of an editing team to establish such conventions. Viewing format and rule definitions as located in the work space, articulation concerns the coordination of the definition work. Such articulation may involve coordination of meetings, ratification procedures etc. So, what one choses to consider the work space determines what articulation is and what not.
A second complication concerns the notations and their interpretations. A notation can only express a message, but no interpretation. An order form, for example, only has meaning when it is interpreted by an agent that knows what to do with the form. The process of interpreting the order form and determining what actions are required, is not part of the notation used in the form, but is determined by knowledge of the meaning of the notation. This knowledge can be represented as a model that describes how an interaction message is to be interpreted and what actions should be performed.

2. A Framework for Modelling Cooperation

Introduction

The goal of this chapter is to develop a descriptive and analytical framework that enables the following:

- A better understanding of the processes and mechanisms involved in cooperative work.
- The identification of generic elements in CSCW.
- A methodical approach to the conceptualisation, design and development of new CSCW systems.

In order to achieve these goals concepts and principles developed in the KADS-projects are used. We start by giving an outline of the problem in subsection “The problem”. Next in subsection “The framework”, the core of this section, the definition of models, is presented. Quite often use will be made of examples, though not all of them will be derived from the CSCW context.

The problem

In the COMIC project an inventory has been made of a large number of CSCW systems. Obviously these systems are extremely heterogeneous in terms of functionality, complexity and the amount of "real cooperation" taking place. A satisfactory classification or categorization is difficult to achieve because this variety tends to obscure the forest for the trees. It seems that a more fundamental approach is necessary to tackle the problem of arriving at a coherent framework for classifying CSCW systems. An additional, probably even more beneficial, effect of this could be the availability of more general design principles that could or should guide the development of future CSCW systems.
Of course other efforts have been made to achieve satisfactory classification schemes. Examples are:

The "contents" of the systems.

Malone (1988) makes a distinction between the amount of domain specific knowledge embedded in a system. Systems range from very general (Hypertext, E-mail) with no knowledge to very specific (advanced project management supports systems) as exemplified by knowledge based systems.

The "structural" properties of systems.

Gibbs (1991) bases his classification on the kind of support systems can offer for the software development process. His five types are almost exhaustive, but in practice actual systems can be put in different categories at the same time.

The nature of the interaction in cooperative work.

Lyytinen (1990) starts from a classification of five salient aspects of cooperative work that can be supported by a CSCW system. These aspects are combined with three metaphors which can underlie the design of computerized support systems. Together they form a matrix containing 15 types of domains in which CSCW could be applicable. Existing systems can be located in this matrix.

As proposed in Gustavsson (1993), another approach is to make a clearer distinction between the levels at which we can describe CSCW systems. It seems that there is strong tendency to have an "implementation" perspective on surveyed systems. He suggests a shift towards a "knowledge level" perspective on CSCW systems. This knowledge level enables a more generic approach in which generalities are stressed over the undeniable implementation differences. The situation as regards CSCW systems seems to be somewhat comparable to the expert system field in the beginning of the '80s when actual systems dominated the thinking. Only later more generic approaches emerged that tackled the underlying similarities in the different systems, for example by identifying generic tasks. This led to the development of comprehensive theories about these systems and the knowledge they incorporate, which in turn spawned methodologies for developing these systems.

It seems not unreasonable to follow the same kind of development for CSCW systems. However, in doing this we can benefit from the experience gained in this related field, more specifically the KADS-approach as developed during the last couple of years (see Schreiber et al., 1994). In this chapter we will follow this road. It should be emphasized that we do not intend to "copy" KADS to CSCW systems, there are probably too many differences. What we will try to do is to apply some of the higher level principles used in the KADS methodology to CSCW systems in general and COMIC in particular. These principles are:

• Identification of generic elements
• Decomposition of the domain in separate components or models that address salient aspects

• Multi-level description of (system) features

  • “Conceptualize before you formalize or implement”, that is embark first on a thorough conceptualization before starting to develop formal languages

This approach comes closest to the one explored by Lyytinen (1990), though we also incorporate a description of the process for developing CSCW systems.

The framework

Models

The basic idea underlying our approach is to carve up the domain into a number of distinct but related areas by means of defining different models (see for the comparable approach in KADS, Schreiber et al.,1994). We do not claim that the models described in this section are at the same level of detail or form a reasonably exhaustive set. Further detailing and refined will be necessary in the future. Figure 1 presents a pictorial view of the models and their relations. The relations in figure 1 indicate how each model is transformed into another model by means of methods and techniques. In contrast to the KADS approach we describe these relations as transformations. In KADS the relations are data relations. Thus figure 1 is basically a process model. However, by spelling out the structural properties of the different models a more static data model representation can be achieved.
COMIC Demonstrator prototypes of Computational Mechanisms of Interaction

The general line of reasoning epitomized in figure 1 is as follows:

- Organisations encounter problems in carrying out their work. Some of these problems can be tackled by means of cooperation solutions. Cooperation solutions are arrangements which require some cooperation between (human) agents. The notion of cooperation is difficult. Many different definitions do exist. We will not go into this but will simply assume that cooperation between agents means that they share to a certain extent a common interest in achieving a certain goal. The context model describes the relations between organisation problems and generalized cooperation solutions.

- General cooperation solutions must be operationalized in workable procedures: cooperation methods. However, solutions embodied in methods quite often generate new problems, cooperation problems. The cooperation model contains a high level description of the cooperation method(s) chosen and the cooperation problems that are associated with them.

- Cooperation methods require of course the active presence of agents. They will have to perform tasks which are part of the cooperation process. The salient properties of agents can be described in the agent model.

- CSCW methods can be seen as specific instances of cooperation methods.
that solve at least some of the associated cooperation problems. Basically CSCW methods are instances of more general methods which are characterized by the introduction of a computer as an agent.

The interaction model describes at an abstract level the interaction between the agents. The main structure of the interaction is derived from the chosen CSCW method(s).

- The abstract description of the interaction model is transformed by applying CSCW techniques into a communication model that represents the operations of the CSCW system.

Finally the communication model is implemented in a CSCW system (the actual observable/behavioral system). This transformation is achieved with (specific) programming techniques.

Let us consider the example of conducting a meeting that has been introduced in the previous section. Meetings between members of some organisational body, such as a departmental board or a project steering group, are typical organisational devices to counter a problem of the lack of confidence type. Important decisions are not taken by one individual but through a formal meeting. In many cases, e.g. the departmental meeting of a University department, it also can serve the purpose of solving a lack of knowledge problem. The meeting is a cooperation method that is a solution of the shared responsibility type. Participants share responsibility for the decisions that the meeting takes. However, not always all members are at the physical location where the meeting takes place. The solution creates a new cooperation problem of distributedness. In order to solve this, some kind of video conferencing or comparable CSCW method can be chosen which will be, together with the agents, combined into an interaction model that describes how the participants in the meeting that are geographically dispersed can use the facilities in order to overcome the cooperation problem. An interesting question is of course whether the CSCW method preserves the necessary articulation mechanisms that should be available for conducting a successful meeting. This clearly refers to the notion of articulation space as introduced in the ‘Introduction’ section. By using CSCW techniques the communication model is built that specifies the operations of the CSCW system. Figure 2 illustrates the role of the different models in the meeting example.
The models and transformations involved are described in the next paragraphs.

The context model

This model contains two components:

- a description of the organisational problems to which cooperation may provide a solution
- a description of the cooperation solution type chosen

We will deal with both in turn.

Organisational problems

In Schmidt (1993) quite a lot of attention is paid to organisational problems surrounding cooperative work. From this discussion it seems to emerge that there are a limited number of problems that can be of relevance.

Lack of capacity. This is equivalent to Schmidt’s ‘‘augmenting of capacity’’, where ‘‘augmenting’’ is in fact a solution type. In general his refers to ‘‘a task that would have been infeasible for the actors individually’’ (Schmidt, 1993, p.7). More specifically this lack of capacity can be of different types.

- A quantity problem, that is an actor could accomplish the task but would spent so much time on it that it would extend beyond his/her own life. Examples are major construction works: cathedrals, pyramids etc..
• A complexity problem, that is an actor simply lacks the skills and perspectives to finish the task in a proper way. Examples are intricate jobs requiring a wide field of expertise.

Lack of Confidence. Although one agent could perform the task at hand, there exists a lack of confidence that a single agent will perform the task in an optimal way.

Physical separation. For whatever reasons (for example specific benefits associated with certain locations) an organisation can consist of physically separated departments that perform interconnected tasks.

This list is not meant to be exhaustive, but serves as an approximation that can be used for illustrating the scope and content of the proposed framework. Additional research and conceptualizing will be necessary to make it more complete.

Cooperation solutions

The organisational problems can be and are being countered by applying cooperation solutions. These cooperation solutions should be thought of as very general types of measures that can be taken in an organisation. An initial list of possible cooperation solutions is given below (some of them are derived from Schmidt (1993)):

Sharing of (information) resources

This solution type entails the notion that resources are not uniquely allocated to one agent but must be utilized by many different agents at either the same or different moment(s).

Collective problem solving

The idea behind this type is in Schmidt's words:

.... a cooperative work arrangement arises simply because there is no omniscient and omnipotent agent (Schmidt, 1993, p.9)

Work decomposition

This type is focused on the classical distribution of work approach, or as Schmidt (1993) labels it “Differentiation and combination of specialties”.

Sharing responsibility

In high risk situations it is sometimes advisable to make a body of persons responsible for taking a decision because a negative outcome would be too damaging for a single individual.

Collective assessment
Individual judgments are pooled to even out large differences.

Again, this list is not meant to be complete. The solution types are very general and indicate only a global direction for a solution. Their use lies in the fact that one still has the freedom to arrange the actual solution in a number of different ways by means of cooperation methods.

The cooperation model

The cooperation model consists of two important elements:

- the cooperation method(s) selected to specify the type of cooperation solution
- problems arising from installing a cooperation method in the organisational context.

These will be described in different subsections

Cooperation methods

Cooperation methods are almost equivalent to what Schmidt (1993) calls ‘‘modes of interaction’’, that is they incorporate some (abstract) procedure that mediates between the agents in such a way that a problem can be solved in principle. A list of possible cooperation methods can be found below.

information management: some mediating mechanism is installed that stores and/or makes accessible the necessary information.

negotiation: procedures are put into place that guide and channel the negotiation process.

(democratic) decision making: a mechanism for deciding in a collectivity about a course of action.

activity control: a mechanism that controls and distributes the utilization of a resource over claimants.

information dissemination: a mechanism that actively distributes information and/or knowledge over agents.

mutual critical assessment: a mechanism for exchanging and pooling individual judgments
formation of reciprocal awareness: a mechanism that enables making explicit shared problems and tasks between individuals

confrontation and combination of perspectives: a mechanism that makes it possible to elaborate and combine different perspectives on a certain domain

Each cooperation method will consist of a task decomposition. That is, the method will consist of a number of tasks that have to be carried out in order to achieve the goal of the method. For this use can be made of the notion of the workspace as introduced in the “Introduction” section. The description of the workspace defines the basic tasks that should be part of the cooperation method as well as the elementary pieces of information exchanged between the tasks. It also contains the dynamic aspects of the method: the sequence of tasks in the workspace. Figure 3 gives an example of an elementary description of a cooperation method: formation of reciprocal awareness as used in one of our case studies.

![Figure 3: Description of a cooperation method (formation of reciprocal awareness)](image)

In figure 3 the boxes represent information items and the ovals tasks, while the arrows on the connecting lines show the sequence.

The picture in figure 3 can be thought of as a kind of generic description of a cooperation method. Constructing cooperation models can be supported by making available a library of generic method descriptions, just as there are generic models of expertise in the KADS methodology. Below we will present some initial specifications of generic cooperation methods mentioned earlier in this subsection.

NegotiationFigure 4 contains a description of another cooperation method: negotiation. This method is characterized by the need to reach an agreement. It starts with an issue about which positions can be asserted. Other assertions can be made and differences between assertions and positions have to be identified.
These differences must be assessed in the light of the negotiation spaces of the participants. Some trade off has to take place in order to reach an agreement.

In figure 5 the Democratic decision making cooperation method is depicted. The difference with negotiation can be found in the voting mechanism that calls for different tasks like “select order”, “vote” and “ratify”.

Figure 4: Description of the Negotiation cooperation method

Democratic decision making
Cooperation problems

However, implementing one or more of these cooperation methods will quite often lead to new problems, so-called cooperation problems of which we can identify the following classes:

Distributedness problems

- **distributed skills**: though the necessary skills are present, they are widely distributed (mainly geographically) so that they cannot be employed easily.

- **lack of information**: the agents involved do not have access to information they need to carry out their task. This information could be in the possession of other agents or not available at all.

- **lack of knowledge**: agents do not possess the knowledge to carry out (parts) of their work. In contrast to information we mean here “active” knowledge, i.e. mainly procedural knowledge about “how” to do the work.

Agreement problems

- **difficulty in reaching consensus**: in order to achieve their work the agents have to reach consensus about aspects of their work.

- **difficulty of making decisions**: in order to achieve their work the agents have to make decisions about aspects of their work.

Resource problems
- **resource clashes**: in doing their work the agents need simultaneous access to the same resource(s), however, these do not permit this.

- **complexity of work**: though the agents have sufficient information and knowledge to do their work the interrelations between their work aspects are so complex that they are not able to coordinate their work in an effective way.

Just as the previous lists introduced, this one is only a starting point useful for illustrating the significance of the proposed framework.

One should keep in mind that there is not a necessary one-to-one mapping between cooperation problems and cooperation methods. One can imagine that some rules for linking the two can be helpful in analysing problems that can be attacked by means of CSCW methods. However, an in depth analysis of these links is outside the scope of this document.

The agent model

The agent model describes the capabilities of the agents that can become or are involved in carrying out tasks defined in the cooperation model. This model can be almost the same as the agent model in the CommonKADS approach. In figure 6 a slightly revised version of this model is presented.

![Figure 6: A representation of the structure of the agent model](image-url)
In figure 6 the rounded boxes refer to the components (or entities) that make up the agent model. The dashed line indicate how the components in the agent model are linked to components in other models. Thus the arrow with the annotation ‘performs’ show that an agent performs a task in the interaction model.

For each agent relevant for the cooperation and interaction model an instance of figure 6 should be constructed. For example, in the meeting case the agents identified in figure 7 must be detailed.

![Figure 7: Agents that play a role in a (university department) meeting](image)

The secretary will need different capabilities than the chair, non-voting participants operate under other constraints than voting participants, the chair will perform different tasks in the interaction model. As soon as a computer will play a role it should also be described as an agent.

**The interaction model**

The link between the cooperation model and the interaction model is made through the application of CSCW methods. These CSCW methods can be seen as specific instances of the abstract cooperation methods described in the cooperation model. As such they are expected to solve the cooperation problem(s) associated with the selected cooperation method. In Schmidt’s (1993) terms they are ‘’means of interaction’’. The distinctions made in this and the previous subsection are also present in Schmidt’s analysis:

Since the modes and means of interaction are semantically neutral in the sense that they may be invoked (with different scope) in articulation work in all work domains, these modes and means of interaction should be conceived of as functions to be supported by a CSCW platform.
(Schmidt, 1993, p.20)

Below some CSCW methods are described verbally:

- conferencing
- referendum
- semaphores
- conversation
- protocol enforcement
- moderators
- mail
- bulletin board

There is no claim that this list is exhaustive or even mutually exclusive. Further analysis and research is needed.

The transition from the cooperation to the interaction model requires the following steps:

- the elaboration of the *articulation space* of the cooperation method
- specification of the *actions* of which the CSCW method consists.
- specification of the *information items* exchanged.
- allocating *agents* to actions.

The first step, elaboration of the articulation space is very important because it should be made sure that the CSCW method employed caters not only for tasks in the work space but also for tasks in the articulation space. In figure 8 an example is given of an interaction model for the meeting example.
In figure 2.8 the upper part described the articulation activities in the articulation space, while the lower part the cooperation method represents. As can be seen in figure 8, agents are added and linked to tasks in both spaces.

The building of an interaction model can be supported by making available a library of generic elements that represent actions and information items that can be used in elaborating the articulation space and the work space. These generic structures can be enhanced and modified in the model building process. Such generic elements could be the ones listed below.

**Actions**

The elementary actions that compose a CSCW method could be (examples from figure 8):

- Create (e.g. a statement)
- Receive (e.g. reactions)
- Send (e.g. assignments)
- Retrieve (e.g. previous statements)
- Display (e.g. question)
- Broadcast (e.g. statement contents)
• Store (e.g. previous statements)

*Information items*

This are the elementary pieces of information that are exchanged between actions. A classification of information items can be derived from the speech act approach, for example (examples are from figure 8):

• assertives (e.g. statements)
• directives (e.g. assignments)
• commissives (e.g. reactions)
• expressives (e.g. questions)
• declarators (e.g. request)

*The communication model*

This model contains the architecture and the infrastructure, media, computational techniques, UI etc.. In terms of Schmidt (1993) this are the “mechanisms of interaction”, formalized means of interaction residing in a computer.

If we take up the notion of a mechanism of interaction (MoI) we can state that the communication model should specify the following properties of a mechanism of interaction (the example is taken from the description of the CEDAC board in Srensen (1994):

**visibility**: the MoI must be visible to the involved actors, in the CEDAC example this refers to the location of the board on the factory floor

**medium**: the MoI is embodied in an explicit medium that is not bound to one single actor (the steel board is the medium in the CEDAC case)

**structure**: the MoI has an explicit structure that is understood by the involved agents, the vertical and horizontal line on the CEDAC board and the direction of the arrow represent the structure.

**symbols**: the “tokens” that are used in the communication, in the CEDAC example there are different tokens involved:

• the error types
• the state of the problem
• the cards itself
**topic(s):** the field(s) of work to which the MoI refers, in the CEDAC case mechanical design and process planning.

**state:** the MoI is at any time in a specific state, a snapshot in time of the CEDAC board for example.

**rules, procedures, protocols:** these restrict the admissible operations in the context of the MoI, in the CEDAC example there are different rule types:

- **definition rules:** the definition of the properties of the MoI
- **change rules:** who may change what at which moment, worker can put up cars, the foreman classifies the cards
- **access rules:** who has access to the MoI at what time, workers can put up notes

**links:** MoI’s are frequently linked to other MoI’s, the state of the CEDAC board is taken as an input to the weekly meeting to discuss the work

**The system model**

Of course there is the actual artefact, the CSCW system. This is at the implementation level. The interaction model and the communication model are implemented in code that enables the running of the CSCW system. Of course there is wide variety of programming languages and platforms that can be used for building the system. Selecting one will be most of the time a question which is too strongly context dependent to make any general statements about. We will return to the system model in the next section.

**3. CPTOOL: a CASEtool for designing CSCW applications**

**Introduction**

The framework described in the previous section consists of a number of models that support the conceptual development of a CSCW solution to a business problem that requires the cooperation of a number of agents. A modelling process that follows this framework will result in an informal model of the interaction forms that constitute the solution. In this section we will show how such a model can be used in the design and implementation of a running CSCW application. For this purpose we have developed a demonstrator tool that supports the
conversion of interaction and communication models into an actual application. The design and implementation process is supported by a software (CASE) tool: CPTOOL, a graphical programming tool for rapid prototyping of CSCW applications.

Starting point of the development of CPTOOL was the methodology that logically follows from the modelling framework. This methodology for constructing CSCW applications consists of the following steps:

- Model the organisational context
- Define required tasks and actors
- Define work space objects and information structures
- Define cooperation forms
- Select Mechanisms of Interaction
- Define Media of Interaction
- Transform and augment the models into a multi-actor Object Oriented system using graphical programming
- Test and evaluate the conceptual and UI aspects of the system with users.

CPTOOL is a prototype tool for developing CSCW applications using the models described in the previous section and using the concept of Mechanism of Interaction (MoI) as the basis for a library of building blocks which support the application development. The graphical programming environment was designed to be close to the conceptual models of the framework, in particular the interaction and the communication models. The support for building and using the other models defined in the framework is beyond the scope of this work.

The general approach in developing CPTOOL was to use advanced GUI and Object Oriented techniques to support the fast construction of prototypes of relatively small-scale CSCW applications.
CPTOOL was built on top of XPCE, an object-oriented development environment, and SWI-Prolog (Wielemaker and Anjewierden, 1994). A number of CSCW-oriented primitives were added to XPCE in order to facilitate synchronous and asynchronous multi-user interaction through Unix inter-process and Internet connections. The general architecture of CPTOOL is shown in Figure 9. A GUI toolkit built on top of XPCE provides a number of classes that can easily be configured into graphical editors. One editor supports the definition of MoI’s that can be imported by the application editor. The application editor provides means to define the user interface components and the underlying communication schemata that are derived from the Interaction Model described in the previous section. The application editor has a code generation facility for generating Prolog facts that are used by the runtime system to execute the application. In the following sections we will first describe what the structure of the resulting application is. Then we will discuss the ways in which CPTOOL supports the application design process. Subsequently we will show how some CSCW applications were developed in a number of examples.

Structure of the Application

The typical architecture of a CSCW application generated by CPTOOL consists of the following components:

- An object class for each role in the CSCW application. For example, the roles chair and participant in an electronic meeting system.
- A dialog window and a behaviour model are associated with each role.
- One or more *shared* objects which serve as a communication vehicle between the various actors using the application.
- Initialisation code which establishes the required objects for a particular session with the application.

Each user in the CSCW application is assigned a certain role and communicates with an instance of the object class corresponding to that role. Communication between users is established through *shared objects* either in the workspace (e.g. a shared text buffer which can be edited by multiple users) or in articulation space (through objects that pass messages and events from one user to another). Each role object class has an associated behaviour specification that describes the message passing structure between the local and shared objects. Messages sent to a shared object are broadcasted to all users of that object. Figure 10 represents the structure of the applications generated by CPTOOL in an abstract way.

As an illustration we show how a simple Bulletin Board like application would look like when built by CPTOOL. In the application a number of users (“Participants”) communicate through a shared directory of files containing notes on a certain topic. Figure 11 shows the interaction model for this application. There is only one type of user: “Participant”. Participants can compose notes that can be submitted to a common repository and they can read notes created by other participants. These activities occur in “Work Space”. In addition participants can submit a new note, or they can be notified when new contributions appear and select contributions for further inspection.
In designing this simple application with CPTOOL a MoI was chosen that is based on a shared repository of files. Each user has an up to date list of all contributions on his or her screen. Whenever a new file is added the list of contributions is updated. Figure 12 shows the dialog window generated by CPTOOL: the user has a number of function buttons, a browser that shows the list of notes in the shared directory and an editor to read or write notes. Names of new contributions are automatically generated from the username and a unique number.

Below the code of this simple application is shown. The class “Participant” consists of a number of objects corresponding to the UI objects such as the buttons “Read”, “Submit” and “Compose”, an Editor object which represents EMACS, a directory object which represents the shared directory and a text-item object “Name” which will hold the name of the participant. Figure \vref{participant-behaviour} represents the behaviour patterns of the application. For example, when a participant pushes the “READ” button, the current list of contributions will be displayed in the browser “Contributions”. Double clicking a selection in this browser will load the file into the editor. Pushing the compose button, creating a note and saving/submitting the file will create a new contribution.

The definition of a class is given by the Prolog predicate \texttt{dialog} with two arguments: the name of the class and a list of definitions:

\begin{verbatim}
dialog(participant, <definition_list>).
\end{verbatim}
The definition list has four parts. The first part defines the name of the class to be defined:

\[
<\text{definition_list}> ::= \{ \text{object} := \text{Participant}, <\text{rest}> \}
\]

The second element of the definition list defines the parts that constitute the class. The class “Participant” consists of a number of objects corresponding to the UI objects such as the buttons “Read”, “Submit” and “Compose”, an Editor object which represents EMACS, a directory object which represents the shared directory and a text-item object “Name” which will hold the name of the participant.

\[
\text{parts} ::= \{ \text{Read} := \text{button(read)}, \\
\text{Compose} := \text{button(compose)}, \\
\text{Editor} := \text{editor(@default, 300, 200)}, \\
\text{Submit} := \text{button('submit')}, \\
\text{Contributions} := \text{list_browser(@default, 42, 8)}, \\
\text{Directory} := \text{text_item(directory)}, \\
\text{Name} := \text{text_item(name)} \}
\]

The third and fourth elements of the definition contain definitions of properties of the constituting objects and specify their layout on the screen:

\[
\text{modifications} ::= \\
\text{Contributions} := \{ \text{name} := \text{‘Contributions’}, \text{show_label} := \text{@on}, \\
\text{Directory} := \{ \text{type} := \text{directory}, \text{length} := \text{15}, \\
\text{Name} := \{ \text{selection} := \text{‘Enter your Name’}, \text{length} := \text{15} \} \}
\]

\[
\text{layout} ::= \\
\text{position(Read, point(45, 22)),} \\
\text{position(Compose, point(48, 67)),} \\
\text{position(Editor, point(59, 223)),} \\
\text{position(Submit, point(41, 117)),} \\
\text{position(Contributions, point(262, 85)),} \\
\text{position(Directory, point(26, 166)),} \\
\text{position(Name, point(25, 185))} \}
\]

The final element of the definition list is the specification of the behaviour of the various objects.

\[
\text{behaviour} ::= \\
\text{Submit} := \{ \text{message} := message(@shared_dir, generate_event, submit) \}, \\
\text{@shared_dir} ::= \\
\text{submit} := message(Contributions, members, Directory?selection?files) \\
\text{Compose} := \\
\text{message} := message(@prolog, compose, \\
?{Directory?selection, file, Name?selection}, Editor) \}, \\
\text{Read} := \\
\text{message} := message(Editor, load, 
\]
The first line of the behaviour specification defines the message that will be generated when the submit button is pushed. When this happens a `generate_event` message is sent to the shared directory object. This message will generate an event that all users for which the object `@shared_dir` is defined will receive. The second element of the behaviour specification defines what message must be generated when a `submit` event is received. In this case the browser “Contributions” is updated to reflect the current contents of the directory. The behaviour of the COMPOSE button is simple: when it is pushed a message is sent to Prolog (which is represented as an object in XPCE) to evaluate the predicate `compose` with the username as argument. Compose is a simple procedure which generates a unique name and invokes the editor. The behaviour of the READ button is that it will send a `load` message to the Editor object with the selected file as argument. Double clicking on the filename in the browser will have the same effect (the `open_message` generated by the browser). The user of CPTOOL could write code such as the class definition described above, but normally the user would use a graphical editor to specify the layout and the behaviour. Examples of the use of these editors is given below. The initialisation code required for starting up the application has to be written by the user in Prolog.

Crucial to the CSCW aspects of this simple example is of course the object `@shared_dir`. This object is an instance of a predefined class, `comic_model`, that contains the required primitives for communication between multiple users. The `@shared_dir` object is defined to have one event port: `submit`.

```
new(@shared_dir,comic_model),
send(@shared_dir,event_port,submit).
```

More complex instances of the class `comic_model` can contain shared status variables that generate events when their value is changed.

This simple application -which is equivalent to a straightforward Bulletin Board- illustrates the basic structure of the application. More complex examples will be discussed below.

Applications can run in one of two modes: single process and multiple process mode. In single process mode communication between users at different sites is established using standard X-11 technology. In multiple process mode communication between the processes is maintained using XPCE primitives that employ Unix sockets for maintaining consistency between shared objects and for message passing.
The applications generated by the current version of CPTOOL run mainly in single process mode, since only very simple shared objects are supported by XPCE in multi-process mode.

Designing with CPTOOL

Methodology

The methodology for designing a CSCW application with CPTOOL consists of the following steps:

- Define the object classes required for each role in the application. This is done in the tool by defining dialog windows for each role or through importing dialog specifications from a library.
- Define the shared objects and their send, get and event ports.
- Define the user interface structure for each role through drag-and-drop operations in the dialog editor.
- Define the Behaviour Model for each role. This involves specification of the messages and events that each role is required to generate or to respond to.
- Write the initialisation code required for the CSCW application to run. This initialisation involves the instantiation of the generic role classes for each participant and the linking of the various user objects through process links.

Creating the Role Objects

The idea behind the tool is that one or more users play a certain role in the cooperative process and that these users use the same interface. The CSCW application designer uses the Interaction Model to define the required roles and to construct the various functional objects in the user interface. In the example of a CSCW application for conducting electronic meetings, one would define roles for a chair person and for an ordinary participant. The participant interface would need to contain some medium of receiving and transmitting the work-related messages related to the meeting. In addition each participant would need to be able to convey articulation messages to the other agents. CPTOOL provides a top-level facility to define the different role objects required for the application. Figure 13 shows the top-level window of CPTOOL for importing, creating and editing role objects. In the example two roles were created: chair and member.
The definition of a number of role objects results in the creation of as many dialog screens that allow the application designer to define the user interface and the objects that the user of the application will have access to.

Editing the role windows and behaviours

The screens for each role can be populated either with primitive interaction objects, such as buttons, browsers or editors, or with conglomerate objects corresponding to a mechanism of interaction. Layout of the screen can be done automatically or by the user. The particulars of each object can be edited by the user of CPTOOL.

Through a drag-and-drop technique interface objects are added to the interface of a role. Of course, these objects have -as yet- no relation, they just share the fact that they are part of the same screen. CPTOOL provides the additional facility to specify -in a graphical editor- the interrelations between interface objects. The Behavioural Model is used for this purpose. The BM consists of all objects that are present in the screen and potentially a number of additional objects. Each object in the BM has a number of ports through which messages can be sent or received.

Computational Mol’s

The notion of Mechanism of Interaction (MoI) occurs in two forms in CPTOOL. The first way in which Mol’s are used is through the shared objects. The tool can use a variety of instances of the class comic_model. The current version of CPTOOL contains definitions for shared objects for:

- named event generation (with and without arguments)
- shared directory
- floor control (one named actor has the floor)
- shareable textbuffer (various variants)
- voting
- agenda representation
- participants lists
This library of shareable objects can be extended to include more complex objects such as shareable spreadsheets, forms of various sorts etc.

A second way in which MoI’s play a role in CPTOOL is through computational realisations of partial models of interaction that represent a MoI. For example, the library contains a model of floor control through a chairperson. This MoI defines two roles, a shared floor control object and the interaction mechanisms for requesting the floor, releasing the floor and assigning the floor. Such a MoI can be imported when an application is developed.

Example Application Generated with CPTOOL

This example shows how the interaction model of a chaired electronic meeting can be implemented using the MOI library and CPTOOL. The model defines two classes of agents: chair and participant. In the workspace the system should support the composition of a statement that can be read by all other users. The medium chosen is a shared text buffer that can be read and edited by an editor such as EMACS. We chose to use a version of the shared object where only one user can edit the text at one moment in time. Editability, i.e. the possibility to make a statement about a certain topic, is controlled through a floor control MOI: a participant requests the floor and the chair assigns the floor to each user as he or she thinks appropriate.

At the work space level each user has access to one shared object: the shared textbuffer. Each user has a private copy of the EMACS editor that is initialised to be the shared buffer. This object is stored in the shared object WSPACE that also contains an item that indicates the name of the current speaker. In addition to the editor and the WSPACE object, the user’s workspace contains an item that contains the name of the user. Figure 14 shows these components in their user interface form and figure 15 shows the corresponding behavioural model. In the behavioural model we see that an initialisation event will ensure that the editor operates on the shared buffer and that the buffer is only editable when the name of the participant corresponds to that of the user - which will generally not be the case at initialisation time.

In CPTOOL this configuration of objects in the workspace can be abstracted to a compound object shared edit that just has two ports: one to get the name of the participant and one to indicate that a new speaker has been assigned the floor. In essence such an abstract object encapsulates the details of the CSCW mechanisms. In fact a user may decide that a more complex local functionality is required and he or she can replace the abstract object by another one as long as the two essential ports are still part of the substituted object. Also note that users can use the local functionality of their sub-system provided by the objects. If the editor object allows multiple buffers the user can prepare contributions in a private textbuffer which can be copied to the shared buffer when the user has control over the floor.
In addition to the objects in the workspace means for articulation must be defined. In this example we require a means for a participant to request and to release the floor. A standard MoI from the library can be called upon to provide the necessary objects: two buttons (Request Floor and Release Floor) and a shared object to communicate any events with the appropriate information to other users, in our example the chairperson. Figure 16 shows the user interface elements corresponding to these articulation functions. The corresponding Behavioural Model is shown in figure 17. In this diagram two objects correspond with the buttons and they convey messages -with the user ‘s name as argument- to the shared articulation object ASPACE. Pushing a button will generate an event that is broadcasted to all other users.

In our simple example only the chair will react to these events, but other users may decide to adapt their system so that they are aware of events generated by other users.

The model for the chairperson agent is somewhat more complex. At the workspace level the chair has to provide the actual text buffer that is going to be shared by all users. This is done at initialisation time. Otherwise the workspace objects and their behaviour are identical to those of the participants. At the articulation level the chair has to maintain a list of floor requests. This list is maintained in a list browser object. Whenever a request floor event occurs the name of the participant requesting the floor will be added to the list. A release floor event will eliminate the participant from the list. When the floor is released the chair can select a new participant from the list to assign the floor to. The user interface objects and their corresponding behaviour come with the MoI library component for chair-controlled floor assignment. Assigning the floor to a new participant will generate an event change speaker that all users receive, and that will determine the editability of the shared text buffer. The behavioural model of the chair subsystem is shown in figure X18.

Sofar we have defined the subsystems for the different roles in the CSCW application. CPTOOL generates code for the corresponding classes and the shared objects. In addition we have to define an initialisation procedure to start the meeting. A simple way of doing so is to provide the chair with a list of users that are logged in and force a window on them. If the user acknowledges willingness to participate in the meeting he or she gets added to the list of participating users and an instance of the participants class gets assigned to them. Currently CPTOOL doesnot provide support to program the initialisation procedures in the same way as the classes for the various roles can be defined.

The application for electronic conferencing described above is quite simple, but it already shows the power of the modelling concepts described in the previous chapters. The application also hints at many variations of the rules of the game that can be thought of. For example, we could easily add the capability of the chair to abort the making of a statement by a participant. We could add abort buttons to each user interface to indicate to the chair that a statement is taking too long. An agenda could be added etc.
Figure 19 shows an electronic conferencing application that has some of these more advanced functions.

4. Conclusions

In this chapter we have described a modelling framework for developing CSCW applications taking multiple viewpoints into account. Each of these viewpoints has an associated model that is used to analyse and design a CSCW application both in its wider organisational context and in terms of the cooperation methods and its Mechanisms of Interaction.

The interaction model and the communication model turned out to be a good starting point for design and implementation of an application. A Computer Aided tool for designing CSCW applications, such as CPTOOL, can make the transition from the -informal- models to a design and implementation relatively easy, as it should be in structured design methodologies. The concept of Mechanism of Interaction turned out to be useful: it plays a crucial role in the development of the interaction and communication models, and it forms the basis for construction of a library of generic components of CSCW systems that support reusability. The current realisation of CPTOOL is only a small-scale demonstrator. Its purpose is to show how the modelling framework can be supported by application generation tools, but its current functionality and content of the library of MoI's are limited.

5. References


Part II: Demonstrator prototypes
Chapter 10

An executable specification of the agent architecture for Coordination Mechanisms

Abstract. The paper presents the main features of the demonstrator under development, whose main goal 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 C2M according to the requirements stated in the field studies.

Monica Divitini, Paolo Evi
DSI - University of Milano

1. Introduction

The implementation of Ariadne is proceeding following two main streams, with the aim of refining both the notation and the agents architecture.

A first implementative effort has been devoted to the formal specification of OAWs and their relationships in OBJSA using the environment ONE (OBJSA Net Environment) (Battiston et al., 1988).

The OBJSA model realized in this way does not only define in a formal way the generic C2M, but it represents an executable specification too. In fact, one of the module made available by the ONE environment is SIM, devoted to the simulation of OBJSA components. This module has been used for the simulation of the modelled C2Ms.

At the same time, in order to obtain a more flexible environment defined at the appropriate semantic level of articulation work, we developed a system for the definition and the activation of C2Ms. Using this system it is possible to define C2Ms based on different relational structures and then verify their behaviour without building up the OBJSA model. The user is driven in every step by a user-friendly interface. The designed environment has been tailored on the requirements and the primitives introduced in chapter 1 and 2, and it allows to work at the three levels of the notation. The system has been defined as an extention of DesignML, a graphical environment for designing nets.
2. Modelling C2Ms with OBJSA

OBJSA nets are a class of modular algebraic high level nets that results from the integration of Superposed Automata (SA) nets (De Cindio et al., 1982) and of the algebraic specification language OBJ (Goguen and Winkler, 1988). They stress the possibility of building a system model through composition of its (sequential no-deterministic) components and encourage the incremental development of the specification and its reusability.

For their characteristics, OBJSA has proved to be a powerful tool for analyzing both the behaviour of the single OAW and the interactions among them. Moreover, the formal specification with OBJSA allowed us to verify the adequacy of the Interoperability Language.

Thanks to the modularity, it has been possible to specify each OAW as an OBJSA component in an incremental way, considering the interactions with the other OAWs one at a time. The compositionality of OBJSA has allowed to use the single components to define the whole C2M, considering the linking among different C2M.

For the development, the composition and the simulation of OBJSA components, the ONE environment is available. Using the simulation module of ONE it has been possible to explore the behaviour of the model and its correctness.

3. Ariadne: the demonstrator

In the following we will try to explain some of the main features of Ariadne, describing them through the interaction with users.

<table>
<thead>
<tr>
<th>ARIADNE</th>
<th>DESIGNML</th>
<th>DRAW</th>
<th>OAWs</th>
<th>ACTIVE ARTEFACT</th>
<th>GRAMMAR</th>
<th>PROTOCOL</th>
<th>INSTANCE</th>
</tr>
</thead>
</table>

Figure 9.3.1 the main menu of Ariadne

The first item (DESIGNML) allows to enter the standard graphical environment offered by DESIGNML. The last three items can be used by the user to specify at which level of the notation s/he wants to work. Selecting the GRAMMAR item the user can select one of the primitives that can be used to act on grammar. The same happens when the user selects PROTOCOL or INSTANCE. The item DRAW allows to enter in a graphical environment where it is possible to design the different proctors. The item OAWs is used to specify the Objects of Articulation Work. The communicative behaviour of these objects has been defined a priori, as illustrated in chapter 2. The user can select one of the objects of articulation work made available by Ariadne and then specify it at the
appropriate level. The item \textbf{AA} allows the user to define the data frame of the Active Artefact and its behaviour specifying the awareness and coordination attributes, as illustrated in chapter 2. The data-frame can be specified by the user as a couple <variable name, variable value>.

3.1. The grammar level (\(\gamma\))

When the user chooses to act at the \(\gamma\) level, s/he can act on grammars either for creation or modification, selecting one of the two items under grammar in the main menu.

When the user wants to create a new grammar, as the first step, s/he is required to insert a name for the identification of the grammar (the system checks for homonyms) and the relational structure on which the grammar must be based. For the moment the structures that are under consideration are labelled graph and SA-Net.

Depending on the structure chosen, the system will tailor its interactions with the user in order to get all the information that are required for the complete definition of the grammar, i.e. the labelling functions and the constraints. During this phase the user is highly driven by the system.

If the grammar that the user is defining is based on graphs, the system will ask the user for the co-domain of the labelling function for arcs and nodes. In the case of SA-net the user must specify the type of the components, and the co-domain of the labelling function for transitions and nodes (figure 9.3.2). in both cases, the system summarizes the acquired information.

(a) the user must choose between Actor and Role the possible name of the components of the net.
The possibility to modify a grammar has not been fully implemented yet. Anyhow, in its final version the system will allow to perform different types of modification. When modifying a grammar, the user must decide, as the first thing, to maintain the existing grammar or not, impacting on the other two levels. The situations that can arise are:

1) The old and the new grammar can coexist. In this case, the user decide to start the definition of a new grammar from an old one, as it is a simpler situation to deal with. However s/he does not intend to cancel the old situation.
2) the new grammar will substitute the old one, but only for future use. This means that the protocols that are already in use are preserved, but that new instances cannot be created.

3) the new grammar must substitute the old one, deleting the existing protocols and related instances.

If the user choose 2) or 3) users will no longer be allowed to use the old grammar, i.e. create new protocol or create instances of protocols that have been previously generated using the old grammar. At this point the system asks the user for the modification that s/he wants to perform on the grammar. The user can perform modifications on the underlying structure, on the labelling functions, on the semantics.

If the user decide to make a structural modification, the system asks for the new relational structure. At this point, the situation is the same that for the creation of a new grammar.

If the user decide to modify the labelling function only, depending on the relational structure on which the grammar is based, the system asks the required modification.

For example, if the grammar is based on graphs, the user can decide to change the co-domain of the function labelling nodes and/ or the one labelling arcs. For instance, if in a given grammar the accepted labels for nodes are Tasks and Activities, the user can decide to cancel one of the given OAW, or, for example, to add Actions to the set of possible labels .

3.2. The Protocol Level (β)

When the user chooses to act at the β level, s/he can act on protocol either for creation, modification, or simulation, selecting one of the three items under protocol in the main menu1.

Selecting the item define, the user can create a new protocol, based on an existing grammar. The system asks for a name to assign to the protocol (checking for homonyms) and it lists the existing grammars, so that the user can select the one that s/he wants to use in order to generate the protocol.

After the selection of the grammar, the user, using the item draw (figure 9.3.3) of the main menu, enters the graphical environment where s/he can design the desired protocol.

1 For the moment, only the item define has been activated.
Figure 9.3.3 The items associated to the item **DRAW**

When the user designs a SA-net, s/he must define the different component. For sake of clarity, each component is in a different page.

When the user design a protocol, the system checks that the protocol is generated following the grammar. While the graphical environment offered by DesignML has been developed for the design of generic nets, the one offered by Ariadne is structured for the design of graphs and nets at the appropriate semantic level.

When the user design the protocol, s/he must specialize the OAWs with respect to the field of work (item **OAW** of the main menu) and define the active artefacts of the C2M (item **ActiveArtefact**).

### 3.3 The Instance Level (α)

At the instance level, the user can decide to create, modify or activate an instance.

Figure 9.3.4 - the items associated to **INSTANCE**

If the user decides to create an instance, s/he must name the instance (the system checks for homonyms) and select one of the existing protocols. The system, depending on the protocol, drives the user to the specification of the
OAWs at the α level. After the definition of an instance, the user can decide to activate it, specifying, for each role, which is the actor that plays that role. At this point, all the components of the instance are automatically recalled by the system and the graphical representation of the activation can start. Starting from the initial marking, the user is presented with a list of all the transitions that are authorized to fire. The offered possibilities are comprehensive of the transitions that are linked to the relation among tasks, activities and actions. In fact, when designing a proctor, the user explicitly specify the interactions with the Active Artefact and with other proctors. A different type of communication automatically managed by the system is the one with tasks, activities and actions.

When a task can start, the system asks to the user to choose the activity.

Depending on the chosen activity, the system is able, on the base of the information contained in the related OAW, to present the actions that compose it, allowing the delegation. In this case, the system is able to maintain the representation of the involved actors coherent with the new situation.

An activation instance can be seen as a set of operation that can be executed by different proctor. We can suppose that each proctor can handle different processes simultaneously. This corresponds to the definition of a new activation instance.

The item **TOKEN GENERATION** allows to define a new activation instance. In other words, a new activation can start without stopping the current ones. For instance, if we consider the Bug Handling example, it is possible to handle at the same time the processing of multiple bugs.

Each activation instance is identified by a univoc number. The evolution of instances is fully controlled by the user that specifies the transition that must fire.

When a transition fires, the corresponding semantics is activated and the local data structures are updated accordingly.

In every moment the user can suspend the activation and query the system on the actual status of the C2M. The user can get information on the Active Artefact and on the proctors' local data structures. This can be done for each activation instance.

The activation of a C2M is very flexible thanks to the enforce function. This function allows to modify the values of the local data structures of proctors and of the data frame of the Active Artefact. It also permits to change the flow of the activation by moving the tokens from one place to another.

4. The graphical environment and the language

DesignML is a highly interactive program development environment. The principal component of DesignML are a high level graphics package called
GRAM and a general purpose programming language called StandardML (Wikstrom, 1987; Sannella and Tarlecki, 1991).

The GRAM (GRaphical Abstract Machine) is a high-level graphics engine which works at the level of connected graphs; in contrast with systems which work at the pixel level. The GRAM has, built into it, a fixed user-interface and behavior pattern. The GRAM however is a programmable machine. It can be programmed to present a different user-interface and to behave differently.

Design ML is a special GRAM application in that it is a development environment for GRAM programs. The DesignML application allows its users to modify the "look and feel" of the GRAM interactively and also provide a facility to generate a GRAM program. Development of GRAM program is carried out in the programming language StandardML which is a very high level, typed, functional programming language. Included as primitive operations in the language are a large number of functions which deal with the GRAM(1990).

Among the other, we recall the operations to install user defined callbacks (i.e. a behavior defined by the user); to create, modify and read diagram structure; to manage the menus.

4.1. StandardML

StandardML consists of two sub-languages: the StandardML "core language" and the StandardML "module language". The core language provides constructs for programming "in the small" by defining a collection of types and values of those types. The module language provides constructs for programming "in the large" by defining and combining a number of self-contained program units.

The StandardML core language is a strongly typed functional programming language. It has a flexible type system including polymorphic types, disjoint union and (higher order) function type, and user defined abstract and concrete types.

The StandardML module language provides mechanisms which allow large StandardML programs to be structured into self-contained program units with explicitly specified interfaces. Under this scheme, interfaces (called signatures) and their implementations (called structures) are defined separately. Every structure has a signature which gives the name of the types and values defined in the structure. Structures may be built on top of existing structures, so each one is actually a hierarchy of structures, and this is reflected in its signature. Functors are "parametrized" structures; the application of a functor to a structure yields a structure. A functor has an input signature describing structures to which it may be applied and an output signature describing which results from such an application.

Signature serves both to impose constraints on the bodies of structures/ functors and to restrict the information which is made available externally about the types and functions which are defined in structure/ functor bodies. Only the information which is explicitly recorded in the signature(s) of a structure/ functor
is available externally. This is vital to allow parts of a large software system to be
developed and maintained independently.

Acknowledgments

The authors want to thank Carla Simone, under whose supervision the demonstrator has been
developed. A special thank is due to the students that have been working at the development of
Ariadne, Monica Corigliano, Daniela Soggia and Claudia Vedruccio. We are indebted to Eugenio
Battiston and Fiorella De Cindio who provided us with the ONE system.

References

Battiston, E., F. De Cindio, and G. Mauri (1988): OBJSA nets: a class of high level netshaving
Verlag, pp. 20-43.
De Cindio, Fiorella , Giorgio De Michelis, Lucia Pomello, and Carla Simone (1982): Superposed
Berlin, Germany: Springer-Verlag.
specification of Abstract Data Types, Wusterhausen, GDR.
This chapter provides a brief technical overview of the Aleph prototype. Three aspects of the Aleph architecture are worth noticeable: (a) the Aleph Architecture, (b) the Aleph User Interface, and (c) the Aleph Language. The Aleph Architecture has been designed with large scale issues in mind. One concern has been to look at the relevance of a Resource Management as a function to support cooperative work in an organisational setting, and the concept of federation for the extension to an inter-organisational environment. The Aleph User Interface provides services for the interaction of people with several interfaces: (1) the Aleph-Tcl language interpreter, (2) the World-Wide-Web interface (3) the X-Windows interface, and (4) the UNIX shell interface. The Aleph Language is a notation based on the Tcl language to monitor, configure and re-specify the computational environment.

1. Description of the Technical Environment

Aleph (ℵ) is a computational environment to support a large scale inter-organisational community of people working together and sharing resources across many types of boundaries.

Aleph tries to address the difficulties of (a) working across distributed locations over the current information infrastructure (Internet), (b) supporting work across organisational boundaries, (c) working in a large environment in terms of people, activity and competition for the use of resources.

Three aspects of Aleph are worth noticeable: (a) the Aleph Architecture, (b) the Aleph User Interface, and (c) the Aleph Language.

• The Aleph Architecture has been designed with large scale issues in mind. Our main concern in the early stages of development has been to validate the theories built on the COMIC\(^1\) Project (Comic 92) and (ComicDel 4.2) on the relevance of a Resource Management as a function to support cooperative work in an organisational setting. The Resource Manager provides primitives

---

\(^1\) This work has been partially supported by the European ESPRIT Basic Research Action 6225, the Spanish CICYT 1486/CE (COMIC) and the Generalitat de Catalunya (FI95-3.326).
to configure and re-specified how actors acquire resources to perform their actions. Therefore, it provides primitives to modify the behaviour of applications independently of the state of the field of work. In addition, other components are also included, such as Object Adapters, Finder, Binder, Context Manager, and a Federation component that provides support for the extension of cooperative interactions between previously isolated environments.

• The **Aleph User Interface** provides services for the interaction of people. In order to enhance malleability (i.e. ability to make global and permanent changes to the behaviour of components) and linkability (i.e. combine diverse components to offer a new service) (Simone and Schmidt, 1994) several mechanisms of access are incorporated: (1) the Aleph-Tcl language and command line interface, (2) the World-Wide-Web interface (3) the X-Windows interface (built on the Tcl library Tk, and Motif X Windows), and (4) the UNIX shell interface.

• The **Aleph Language** is a notation to monitor, configure and re-specify the computational environment. Every Aleph component contributes with primitives at the appropriate semantic level, independently of the work going on. The Aleph notation can be used to specify, build or link the salient dimensions of cooperative work. The Aleph notation is based on the Tcl language (Ousterhout, 1994), an interpreted language intended to be the skeleton of a scripting language that can be easily extended with new primitives and incorporated into applications as in our case.

An important feature in Aleph is the support for the establishment of inter-organisational links in an easy manner. We use common Internet protocols such as nntp (rfc977) for asynchronous communication, or mime (rfc1341) to exchange objects.

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 (Benford et al, 1994). The S*S has grown as a result of the work in the COMIC Project 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. It is therefore 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.
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 goal is to move cooperative issues in the user interface out of applications, and it is being implemented as a kind of CSCW Widget set, similar to a X-Motif widget set but able to represent shared interaction coming from different screens, people and applications.

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 on 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).
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 work articulation), event circulation (to propagate information about relevant actions and changes), awareness (knowledge about each other's actions).

This architectural model is addressing the needs of people involved in an organisational CSCW system: application designers, organisational authorities, and workers or users.

For the application designer, S*S support the access and reuse of all different components at any level and select which primitives are going to be combined to specify and design a new application.

For the organisational authority it supports the incorporation of organisational policies and restrictions in components.

For workers or users, prototype implementations of the architecture usually provide control panel tools for every component, and a browser interface to visualise the status of the environment and launch tools.

3. The Aleph Prototype

The Aleph system is a partial implementation of the S*S architectural model. It is intended to be a test bed for the testing and demonstration of the architecture with special emphasis on the issues raised on the issues raised on the COMIC Project regarding large scale social interaction environments.

The S*S Architecture has been partially demonstrated on several prototypes systems (ComicDel 4.2) using diverse distribution support platforms such as a
CORBA compliant distribution mechanism (Group Desk), the MultiGossip distribution package (Cooperative Work Interface Builder, CoBoard), ISIS (ISIS, 1990) (Collaborative Desktop, COLA and Aleph). Even though we started working with ANSA, we moved to ISIS because of the ability to send messages directed to groups of processes.

Figure 3. The Aleph architecture in detail

There are no critical requirements for the communication quality of service, but a basic invocation service able to interact with a group of processes with ordering guarantees, a mechanism for event distribution among distinct processes. This is provided in our prototype by the virtual synchrony model of ISIS toolkit (ISIS, 1990). While ISIS provides a good mechanism for group communication and support for building fault tolerant services, other platforms such as ODP based (ANSA) or OMA/CORBA based have proved to be useful specially in the coming years when the Reference Model for ODP and the Object Management Architecture are mature.

Furthermore, providing fault tolerance has been one of our purposes. It means that the services should be available at any time. We have chosen to replicate all the servers in order to provide the required level of quality of service. ISIS provides mechanisms for that with process groups, group multicast, group RPC and the rexec facilities (remote-execution).

We use the file system to store the data used by the managers. We have chosen this solution because, despite its simplicity, it was enough to guarantee a minimum level of quality. In addition, we use a usenet news store to distribute MIME objects among several environments. Moreover we are considering to link with well-known databases of people and resources such as CSO (Dorner and Pomes, 1992).

A high level of environment accommodation is also needed by our prototype. Our prototype is related with the organizational links and rules of any organizational context, such an enterprise. Any enterprise has a high level of
dynamism and its rules can change every day. So, the system that tries to address these problems has to be adaptable enough to represent these changes. The components of the prototype concerned with managing these rules have been implemented using the Tcl-Tk (Ousterhout, 1994). This language provides the capability of easily modify any component, such as any rule or policy.

Finally, in addition to ISIS we have used the C/C++ basic socket libraries to link several Aleph prototypes working in different environments (a federation).

In the following sections we describe these main ideas in terms of Resource Management and other related functions such as trading, binding, and context management. Afterwards, the mechanisms for external interaction are described. This means interaction with the user, but also interaction with other environments with the purpose of inter-system interaction.

4. Interacting with people

Aleph has various interfaces tailored to various purposes and people. The intention is to provide the appropriate primitives, in terms of interaction style and semantic level for managing, working or building the Aleph environment. Basically there are two different interaction styles:

- command line interface, intended for the construction of new primitives, tools and routine management tasks.
- windows interface, intended for people who work on Aleph, either using tools, sharing information, articulating work, or managing, defining policies, monitoring or configuring a certain domain of work.

There are two command line interfaces. The Shell interface is composed by a set of UNIX commands that may be combined in shell scripts. This interface has proven useful for debugging purposes and for routine housekeeping tasks. Shell scripts may be created to interact from UNIX with the Aleph System.

As an evolution of the previous, the Aleph-Tcl interface and language (Simone and Schmidt, 1994) is a powerful open language that easily incorporates the primitives from the Aleph components to support the construction of new components, mechanisms and applications.

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, 1994).
Figure 4. Aleph-Tcl extends the Tcl language with primitives at the level of 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.

Since it is a simple interpreted language where modifications may be introduced while a script is running (commands sent to a script are executed by the interpreter) and events may also be handled by the interpreter (event may be bound to Tcl functions).

There are two windows interfaces: a X-Windows interface, and a World Wide Web interface.

The X-Windows interface has been built with Tcl-Tk, an extension of Tcl for the construction of Motif based interfaces. This interface provides generic tools such as a browser, and administration tools in form of control panels for each manager or component.

The WWW interface is a http (http) compliant interface that is intended for users who browse through the Aleph system from any location, using any WWW client application. The WWW protocol provides rather limited but appropriate browsing capabilities in our attempt to support large scale distributed groups. Applications would be invoked by using the capability of launching external applications of many WWW clients.

5. Interacting with other environments

Aleph has been designed with large environments in mind. It addresses the problems that occur when two or more systems have to cooperate in order to carry out a certain task. Therefore, scalability and heterogeneity are important issues. By scalability we mean that the system must be able to grow without affecting the current state of the existing systems. Heterogeneity is closely related to the diversity of large social and computational settings (Schmidt and Bannon, 1993).

In order to enable cooperation among a large and heterogeneous set of entities, we have chosen a federated scheme, where entities decide to cooperate keeping
freedom of association and there is no central authority. Little agreement and mutual knowledge between domains is required before interaction can occur partially because every organization is responsible for itself, it will apply their own management policy and restrictions for the environment they own.

After an agreement between two organisations is achieved, and before the federation is operative (some forms of cooperation will then be possible across both organisations), boundaries have to be identified and circumvented with boundary objects. An **Interceptor** is the element that permits to cross these boundaries by bridging the gap between two heterogeneous domains. In Aleph we address this problem with the **Federation managers** and the **Aleph-Tcl news interface**.

The Aleph-Tcl news interface is a mechanism to distribute asynchronously data without stopping their activity. As explained above, all the information of each manager could be represented using the Aleph-Tcl notation.

The Federation manager is the element that controls and manages all the interactions with foreign objects. It will carry out the necessary connections with other entities (other federation managers) in order to resolve a given request. Interaction with other entities will be always hidden from users. In addition, the federation manager is also concerned with the management of interceptors. Therefore, the federation manager provides transparencies across a federated environment: several environments linked by the federation mechanism look as one single environment for some purposes. For instance, a context manager would contact the federation manager to link with a remote context manager. These federation managers have to comply with some communication policies in order to work efficiently.
6. Work in progress

Current work is oriented to the development of better mechanisms for the federation of domains and the use of interceptor objects to cross discontinuities. Linking organisational environments may be very complex in practice. Issues of interest are the definition of a common conceptual and operational framework before contact may take place; security and privacy, and the use of rule based languages to support the negotiation of contracts and the translation of transactions among domains.

References

COMIC (Computer-based Mechanisms of Interaction in Cooperative Work) CEE, ESPRIT Basic Research Project 6225.
Network News Transfer Protocol, URL=http://ds.internic.net/rfc/rfc977.txt
HyperText Transfer Protocol specification (reference moved to www.w3.org instead of info.cern.ch), URL=http://info.cern.ch/hypertext/WWW/Protocols/HTTP/ HTTP2.html
Multipurpose Multimedia Mail Extensions (MIME) (reference moved to www.w3.org)
URL=http://info.cern.ch/hypertext/WWW/ Protocols/rfc1341/1_Introduction.html
