Extending the Scope of PAC-Amodeus to Cooperative Systems

Daniel Salber, Laurence Nigay, Joëlle Coutaz

Laboratoire de Génie Informatique, LGI-IMAG
BP 53, 38041 Grenoble Cedex 9, France
Tel. +33 76 51 48 54, +33 76 51 44 40
E-mail: {Daniel.Salber, Joelle.Coutaz, Laurence.Nigay}@imag.fr

1. INTRODUCTION

Software tools for the construction of interactive systems such as user interface toolkits, application skeletons, and user interface generators, alleviate the activity of programming but do not eliminate software architecture modelling. For example, the "callback" mechanism made popular by X-Window, does not enforce the distinction between domain-specific concepts and presentation-specific issues. Without a framework for prescribing the appropriate usage of callbacks, the software designer may develop an interactive system that cannot be maintained nor modified. Cooperative systems, which draw upon a wide variety of tools, techniques, and constraints, make architecture modelling even more crucial.

In order to reason about software modelling for cooperative systems, we first need to clarify the nature of the problem space. The role of a problem space is to identify the set of concepts relevant to a particular domain of interest and to organize these concepts into a framework suitable for rising the appropriate questions, for clarifying the scope of the solution space, or for characterizing a particular design solution. The litterature reveals a wide variety of issues related to CSCW but these concepts are not organized in a way that supports software architecture modelling. We claim that one goal of the workshop is to define a problem space useful for reasoning about software architecture and use this framework as foundations for comparing, devising, and assessing current and future architecture models.

In section 2, we present a draft proposal of the problem space for the software design of cooperative systems. In section 3, we describe PAC-Amodeus, our own conceptual architecture model, and show how it can support the issues made explicit in our problem space.

* ACM 1994 Conference on Computer Supported Cooperative Work, "Software Architectures for Cooperative Systems" Workshop, October 22-26, 1994, Chapel Hill, North Carolina, USA.

2. A PROBLEM SPACE FOR THE SOFTWARE DESIGN OF CSCW

Figure 1 shows our early attempt to organize the concepts related to CSCW into a problem space useful for modelling the architecture of cooperative systems. We do not claim exhaustiveness but propose a starting point for discussion and refinement during the workshop.

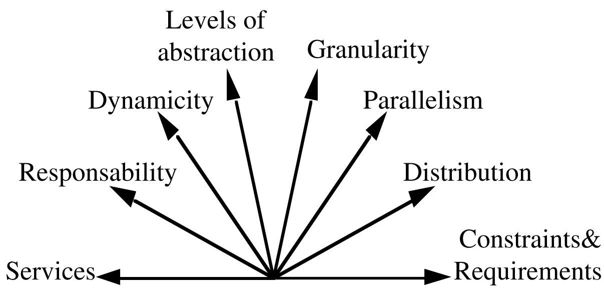

Figure 1: Problem space for the software design of CSCW.

The very first dimension to consider covers the *services* that cooperative systems are able or should be able to support. These are, but are not limited to, access control and authentification, quality control over the communication media, communication mechanisms and policies, and group support. Clearly, access control, quality control, and so on, are flag names that must be refined appropriately. In particular, "group support" encapsulates the notions of user, role, group memory, etc.

The second dimension expresses the *responsibility* for ensuring a particular service. The service (say access control), may be performed by the users themselves, not by the system. Timbuktu's shared-mouse mode [Farallon 87] is a typical example of "social control". Alternatively, responsibility for the service may be shared between the users and the system, or may be performed by several components of the system at some level of abstraction. In the Cavecat mediaspace [Mantei 91], the definition of the connection rights is shared between the users and the system: users are in charge of setting their level of availability whereas the system decides which connection types are allowed for a given level of availability. In some communication systems such as Unix's talk command, connection acceptance is performed socially (e.g., on the fly reject) while in others such as in Rave [Gaver 92] acceptance is purely technical.

Dynamicity covers the capacity of the system to support changes during a session. For example, for a given service (say role assignment and access control), it expresses the capacity of the system to support dynamic changes in role assignment and access control. Coupled with the responsibility axis, it means that responsibility for modification may (or may not) migrate dynamically between the users, or between the users and the system, or even between the components of the system at some level of abstraction. For example, in Timbuktu, floor control can migrate between the participants. In Cavecat responsibility for connection acceptance is shared but is static. In EuroCODE [Bellotti 93], a user “A” may observe that another participant “B” is busy but will accept glance connections from “A”. Although authorized by “B”, “A” may decide not to glance at “B”. Because participants availability is made observable by the system, connection control can migrate dynamically between the technical and the social modes.

Services, such as access control and communication mechanism, may be performed at multiple *levels of abstraction* ranging from the facilities provided by the operating system to special purpose policies implemented in the functional core. Others, such as device independence, may involve a very specific level. The levels of abstraction of the Arch model [UIMS 92] augmented by the underlying operating system, provide a possible basis for structuring this axis. Clearly, depending on the stage in the reification process, the software designer may need to refine the operating system slot into the set of abstract machines at hand (typically, distributed object oriented systems such as GUIDE [Krakowiak 90] offer higher level services than traditional multiprocess systems such as Unix). Coupled with responsibility and dynamicity, one can express, for a particular service, the capacity of migration between the levels of abstraction of the system. Thus, migration may occur among the actors of the services (as discussed above) as well as between the levels of abstraction of the software components.

Granularity is also a general issue: “How big is a chunk?” [Simon 84]. Applied to group support, it expresses the size of the group. Coupled with dynamicity, we cover the dynamic change of the number of participants in a session. (If so, the system must include a service for group memory such that newcomers can get to the current state of the task.) Granularity is also relevant to access control: at some level of abstraction, what is the coverage of the protection mechanism? For example, in an object-oriented operating system, does sharing use objects as the unitary scope, or does it use a finer grain such as object methods? In a cooperative system like MMM [Bier 92], sharing at the method level can facilitate the implementation of functions that allow multiple users to modify the same object synchronously: while one user changes the color of a rectangle, another one can resize the same rectangle or two users may modify the size of the rectangle using opposite corners. In the latter case, the same method is activated simultaneously but with distinct parameters.

Parallelism is a general phenomenon that is not specific to CSCW but useful to express the temporal relationships of the activities performed by the actors of the task (i.e., the users and the software components of the system). When considering the users, it expresses the capacity of synchronous or asynchronous collaboration or, when dynamic, the variation between these modes which may be unpredictable [Grudin 94]. As for the software components, one can refer to the description of the MSM framework we have developed for multimodal interaction [Coutaz 93].

The actors are characterized by their topological *distribution*. For users, one may ask whether they are in the same room or not, or a mixture of both as in teleconferencing. Coupled with dynamicity, distribution expresses the capacity of users to change location as in ubiquitous computing. Location may thus be unpredictable [Grudin 94]. The distribution of software components over the network covers the litterature on software architecture for distributed systems including the two extremes, centralized vs. replicated. Coupled with dynamicity, location may vary over time. As observed by Bentley et al, [Bentley 94], architecture for CSCW sits between the two classical extremes. We do support this idea and claim that the “slinky” metaphor used for the distribution of semantics across the levels of abstraction of a system applies as well to the distribution of the location of the system components. As for semantics, distributing code relies on careful and experienced engineering tradeoffs. Constraints and requirements provide the driving force for selecting informed trade-offs.

Constraints and requirements are setup early in the development process. They cover the description of the hardware and software tools used for implementing the system. For example, the underlying infrastructure may be heterogenous and may show variability in performance. Requirements include the quality plan which specifies the factors and properties that the system should satisfy. In addition to software engineering factors such as portability and reliability, we must recruite HCI-centered criteria such as response time stability, observability, honesty and other general properties such as those presented in [Abowd 92]. All of them may have a direct impact on the nature of the services to be provided, on their distribution, on their level of abstraction, etc. For example, if response time stability is required, one may want to explore replication of the service augmented with semantic delegation (capacity of the user interface to handle domain-dependent functions and concepts).

3. PAC-AMODEUS AND ITS INTERPRETATION TO SUPPORT CSCW SOFTWARE DESIGN

3.1 PRINCIPLES OF PAC-AMODEUS

PAC-Amodeus [Nigay 91] blends together the Arch and PAC [Coutaz 87] conceptual models. The Arch model has the nice feature of considering engineering reality such as trade-offs between software criteria and the constraints imposed by implementation tools (e.g., Motif) or the pre-existence of a functional core. The five components structure of Arch includes two component adapters which allow the software designer to insulate the key component of the user interface (i.e., the Dialogue Controller) from the variations of the functional core and implementation tools.

The Arch model however, does not provide any guidance about the decomposition of the Dialogue Controller nor does it indicate how features such as parallelism can be supported within the architecture. PAC, on the other hand, stresses the recursive decomposition of the dialogue controller, in terms of agents, but does not pay attention to engineering issues such as the existence of implementation tools. PAC-Amodeus gathers the best of the two worlds. Figure 2 shows the resulting structure.

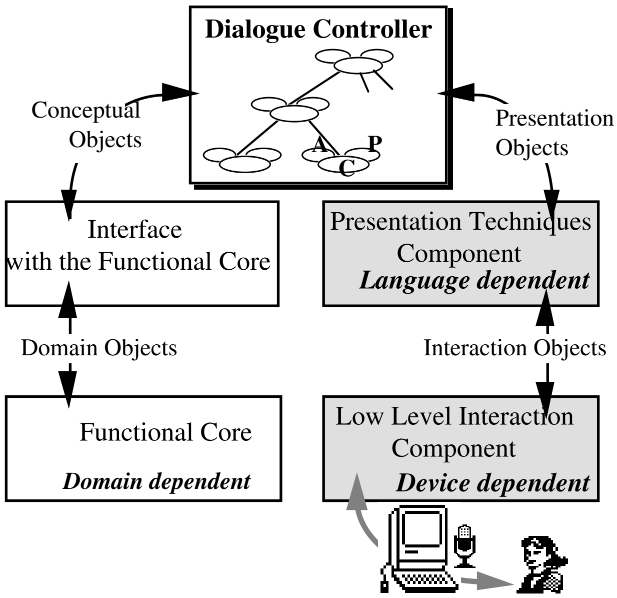

Figure 2: The PAC-Amodeus software architecture model.

PAC-Amodeus goes one step further than Arch in two ways:
(1) it makes explicit the boundaries of the Presentation Techniques and the Low Level Interaction components using the notions of *physical device* and *interaction language* as defined in [Nigay 94]. (A device, e.g., a keyboard and a screen, is an artefact that acquires or delivers information. An interaction language such as a pseudo natural language and direct manipulation languages defines a set of well-formed expressions that convey meaning.) The Low Level Interaction Component is necessarily device dependent and language dependent; the Presentation Techniques Component is device independent but language dependent; the other components of the interactive system, including the Dialogue Controller, are both device and language independent
(2) PAC-Amodeus decomposes the Dialogue Controller into a set of co-operative PAC agents. This set is derived for a particular system using the heuristic rules presented in [Nigay 94]. In a nutshell, a cluster of agents is used to support a user's task. Interleaved and parallel tasks derived by a task analysis, are modelled as clusters of co-operating agents. Clusters of hierarchical agents are convenient ways of modelling the *abstraction/rendering functions* mentioned in the MSM framework [Coutaz 93].

3.2 APPLYING PAC-AMODEUS TO COOPERATIVE SYSTEMS

The PAC-Amodeus model can then be extended to be applied to cooperative systems. A straightforward way is to replicate the arch for each user. But this approach may lead to unnecessary duplication of software components and it doesn't make explicit the communication between components at the same level nor the sharing of the components between the users. The framework introduced in section 2 provides valuable input to decide which components should be shared (or not) and which components should communicate with each other at a given level of the arch. Basically there are three ways a given level of the PAC-Amodeus model can be arranged: common, coupled, or decoupled as shown in figure 3.

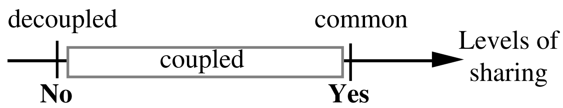

Figure 3: Sharing a PAC-Amodeus component.

A component of the model is put in *common* when it is shared by multiple users. This is the case of the functional core component for collaborative editors: all users manipulate the same common document and the functional core, which handles access to the document can thus be shared. This is the approach that has been adopted by ALV [Hill 92]. However, other components such as the Low Level Interaction Component (LLIC) at the other end of the arch can also be shared: in Timbuktu's shared-mouse mode, users share the same pointer. Since the LLIC handles device-level interaction, putting the LLIC in common is a convenient way to achieve device-sharing.
It is important to note that when a component is common in the conceptual architecture, it does not necessarily mean that there is a unique component in the software implementation. Although this is a reasonable approach for the functional core (this is the classical centralized CSCW model), performance reasons would forbid this for the LLIC. Actually a common LLIC in the model may mean that the LLIC is replicated on each user's machine and that the LLICs communicate to maintain consistency between themselves. These implementation issues are also very dependent on the underlying operating system support (e.g., distributed operating system, communication facilities) as expressed by the *constraints and requirements* axis of the framework of section 2.

Components in a given level of the PAC-Amodeus model are *coupled* when they have to maintain some sort of consistency. The typical case is the necessity for feedthrough, i.e., each user must be aware of the other users' actions. For example in the SASSE collaborative text editor [Baecker 92], each user is made aware of the position of the other users in the document through a set of scrollbars. Each scrollbar reflects the position of another user. However, the scrollbars are not shared for input and each user may only manipulate his own scrollbar. This is a case where a component is not common as in the above examples, but there is some coupling between the software components that are in charge of the scrollbars. Any change in one of the scrollbars is reflected in the other scrollbar-handling components. The coupled components maintain a looser consistency than common components.

Lastly, components for different users may be decoupled. This is the simple case where there is no sharing. At a given level of the PAC-Amodeus architecture, the components are distinct; they function independently and there is no communication between them. For example, in a simple collaborative drawing editor where all users work on the same document but there is no other sharing, the LLICs will be decoupled: each user will have his own local copy of the LLIC.

These three different levels of sharing happen at a global level in the software architecture. If we go into the details of the sharing of a component, we can discover that there are many different ways to share a component: there are many possible combinations to link the set of inputs to the set of outputs. These possible combinations are still an area of investigation.

4. ACKNOWLEDGMENT

This work has been supported by project ESPRIT BR 7040 Amodeus 2 and by PRC Communication Homme-Machine.

5. REFERENCES

[Abowd 92] G. Abowd, J. Coutaz and L. Nigay. Structuring the Space of Interactive System Properties, in Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction, Ellivuori, Finland, pp. 113-128.

[Baecker 92] R. M. Baecker, D. Nastos, L. R. Posner, L. K. Mawlby. The user-centered iterative design of collaborative writing software, in *Proceedings of the Workshop on Real Time Group Drawing and Writing Tools*, CSCW'92, Toronto, 1992.

[Bellotti 93] V. Bellotti, EuroCODE AV Exemplar, Amodeus Esprit Project document ID/IR 6, 1993.

[Bentley 94] R. Bentley, T. Rodden, P. Sawyer, I. Sommerville. *Architectural Support for Cooperative Multiuser Interfaces*, IEEE Computer, May 1994.

[Bier 92] E. A. Bier and S. Freeman. MMM: A user interface architecture for shared editors on a single screen, in *Proceedings of the UIST'91 Symposium on User Interface Software and Technology*, pp. 79-86.

[Coutaz 87] J. Coutaz. PAC, an Implemention Model for Dialog Design, in *Proceedings of Interact'87*, Stuttgart, September, 1987, pp. 431-436.

[Coutaz 93] J. Coutaz, L. Nigay, D. Salber. The MSM Framework: A Design Space for Multi-Sensori-Motor Systems, in *Proceedings of the East-West HCI'93 Conference*, Moscow, 1993.

[Farallon 87] Farallon Computing Inc. *Timbuktu User's Guide*, 1987.

[Gaver 92] William Gaver, Thomas Moran, Allan MacLean, Lennart Lövstrand, Paul Dourish, Kathleen Carter, William Buxton. Realizing a Video Environment: EuroPARC's Rave System, in *Proceedings of the SIGCHI'92 Conference*, ACM Press, pp. 27-35.

[Grudin 94] Jonathan Grudin. *Computer-Supported Cooperative Work: History and Focus*, IEEE Computer, May 1994.

[Hill 92] R. D. Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications, in *Proceedings of the SIGCHI'92 Conference*, ACM Press, pp. 335-342.

[Krakowiak 90] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin. *Design and Implementation of an object-oriented strongly typed language for distributed applications*, Journal of Object-Oriented Programming, September 1990.

[Mantei 91] M. M. Mantei, R. M. Baecker, A. J. Sellen, W. A. S. Buxton, T. Milligan. Experiences in the Use of a Media Space, in *Proceedings of the SIGCHI'91 Conference*, ACM Press, pp. 203-208.

[Nigay 91] L. Nigay and J. Coutaz. Building User Interfaces: Organizing Software Agents, in *Proceedings of the Esprit'91 Conference*, Brussels, November 1991, pp. 707-719.

[Nigay 94] L. Nigay, Ph.D Dissertation, University Joseph Fourier, Grenoble, 1994.

[Simon 84] H.A. Simon, *The Science of the Artificial*, The MIT Press, third edition, 1984.

[UIMS 92] The UIMS Workshop Tool Developers: A Metamodel for the Runtime Architecture of an Interactive System, in *SIGCHI Bulletin*, 24, 1, January 92, pp. 32-37.

6. SLIDE FOR PRESENTATION AT THE WORKSHOP