Formal Specification of a Framework for Groupware Development

Alain Kerbrat, Slim Ben Atallah
Bull-IMAG, 2 avenue de Vignates, 38610 Gières, France E-mail :
`[Alain.Kerbrat,Slim.Benatallah]@imag.fr`

Abstract

This paper describes the formal specification in LOTOS of COOPSCAN, a framework for cooperative applications development. We first present the architectural choices and collaboration strategies for the integration of applications in such a collaboration aware framework. Then, we show how we derive a LOTOS specification from the CoopScan description. We define some generic properties to be verified in this kind of framework. The verification of some of these properties is presented, using the CÆSAR-ALDÉBARAN toolbox.

Keywords: formal specification, verification, LOTOS, CSCW, groupware, software components

1 INTRODUCTION

As local networks and teamwork develop, the concept of Computer Supported Cooperative Works (CSCW) becomes increasingly important. Cooperative applications have been classified following several criteria. One deals with the nature of interactions between participants, either asynchronous or synchronous (every action on the shared space is simultaneously visible by all participants → What You See Is What I See). In this paper, we focus on the synchronous cooperative applications.

The development of such applications can follow two basic approaches : either develop the whole application from scratch, or re-use existing applications and integrate them into a collaboration aware environment. The second approach seems more interesting, and is developed now in several groupware frameworks(e.g. [Ca90], [MR92]). Although these frameworks have different characteristics and fulfill various needs, they all offer a structured view of what is (or should be) a cooperative environment, and how should an application be designed or modified in order to be integrated in such an environment.

However, all the ideas about the architecture and protocols of a synchronous groupware development are generally expressed informally, and there is no way to actually test the consistency of architectural choices, or protocols combination without implementing it.

The aim of this work is to extend an existing development environment with a formal description of its architecture and functionalities. We first intend to specify and verify some generic properties on the described architecture. Then it will be possible to actually analyze and test the integration of different protocols for the management of collaboration.

The rest of the paper is organized as follows : first, we describe the intended framework for groupware. This environment has already been designed and is currently being prototyped. In a second part, we present a component based description of the CoopScan's architecture and how we obtain a Lotos [BB88] specification, then we extend the Lotos specification in order to model a working environment.

In the last part, we define some important properties for cooperative applications, and then show how they can be verified using the Cæsar-Aldébaran toolbox.

2 THE CoopScan FRAMEWORK

CoopScan [BAK95] is a framework for developing CSCW applications. In CoopScan, a clear separation is drawn between the cooperation space, which contains the shared data and applications, and the organization space, where the collaboration control protocols are defined. These protocols concern the following points:

The registration protocol controls the actions performed when a user initiates, joins or leaves a collaboration session. The main problems occur when a user joins an existing session, as we have to provide him with an up-to-date copy of the shared data of the session. Furthermore, all participants must be aware of the new comer.

The concurrent access control protocol controls the management of shared data access conflicts. We will consider here a pessimistic policy, where we prevent conflict by associating the access right with an unique token.

2.1 CoopScan architecture : a component based description

We describe the CoopScan architecture using components, as defined in [BBA$^+$95]. Components in this proposition are well encapsulated parts of the application, which communicates with each other via services calls. One strong point of component based descriptions is the clear separation between the description of a component and the description of its interactions with other components. We consider two classes of components :

Basic components These components correspond either to component reducible to one single object or binary module, or to a bunch of objects where we cannot make a clear distinction between the code of the objects methods and the code for the objects communications. In the latter case, we will encapsulate all the set of interacting objects in one single component.

Complex components These are components composed of sub-components, and of a description of all potential interactions between sub-components. In[BBA$^+$95], another characteristic of component is distinguished for the design of cooperative applications ; this special component is the *collection* component, i.e. a component which can exist as several instances at execution time.

Any component offers two distinct, but related views : its interface which *declares what* can be its interactions with its environment, and its implementation which *defines how* it evolves with respect to these interactions and time progress.

Component's interface A component's interface describes all the services that the component provides (offered services) or needs for his computations (required services). The services are described by their signatures, much like the Interface Definition Language used in the CORBA framework [Gro93].

Component's implementation A component's implementation differs in nature if we consider basic or complex component. In the case of a basic component, the implementation can be a binary object, ready to be linked with the rest of the application.

In the case of a complex component, the implementation consists in a list of sub-components (either basic or complex, possibly collections) and a *controller.* The controller first role is to describe all potential communications between sub-components. This is a static view of the communications between sub-components. The controller's second role is to provide also a dynamic view of the communications, by the specification of *ordering sequences* of the communications and of the dynamic creation/destruction of instances of collection sub-components. The controller last role is to keep track of some attributes, either public or private variables, known at the component level.

Graphical description To present the COOPSCAN 's components, we will use a graphical syntax, which comes from the configuration language DARWIN[Dul94]. In this language, a basic component is represented as a named rectangular box, and the services of the interface of the component are drawn as small squares or circles. A empty square or circle means a *required* service, whereas a full one means a *provided* service. A collection component is represented as several superimposed boxes, with the interface services drawn only on the uppermost box. The minimum and maximum number of instances allowed is given by a interval *min ... max* following the name of the component.

2.2 Component CoopScan

The component *CoopScan* is described as a collection box, of max cardinality n which is the maximum number of users. We describe in the same figure the structure of the complex component *Site*, by drawing its two sub-components, *UserSession* and *UserInterface. UserInterface* is the COOPSCAN interface, aimed at the control of the collaboration; it is distinct from the shared application user interface. *UserSession* is a complex component which encapsulates all the components related to the con-

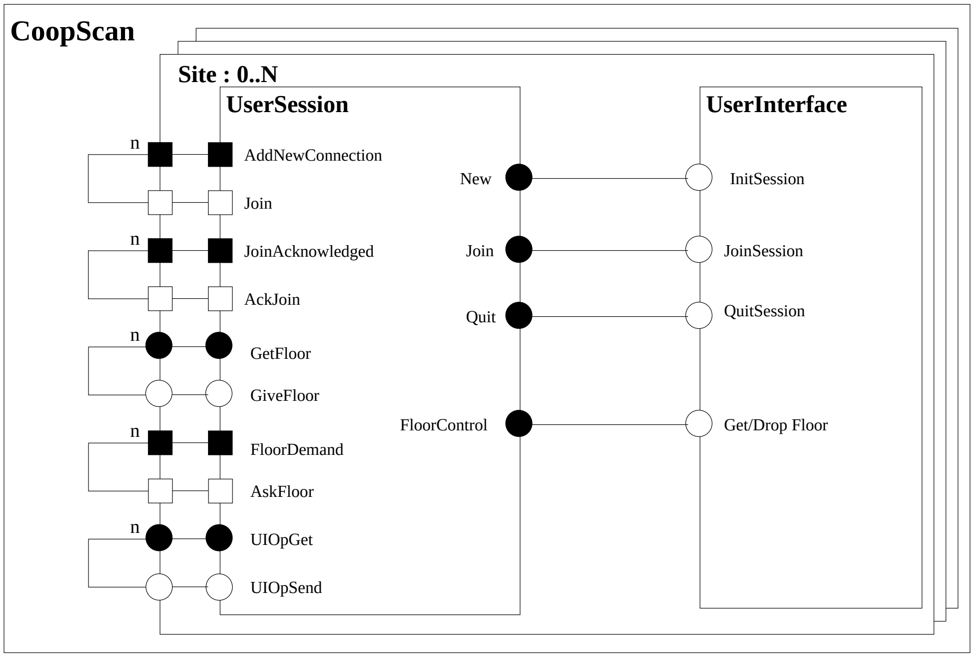

Figure 1 Component *CoopScan*

trol protocols. The complete description includes 3 complex components and 6 basic components.

The use of a collection component at the highest level is highly typical of cooperative applications, when we consider a replicated architecture. A centralized architecture would be described by adding a sub-component corresponding to the shared application and connected to all the instances of the component *Site*.

2.3 From components to LOTOS

The first step of the translation from components to LOTOS is to associate with every basic component an abstraction of its behavior, in terms of the services it presents in its interface. The next step is to translate the description of complex components into a LOTOS specification. The graphical description of the COOPSCAN architecture gives a clear indication on the parallel structure of the LOTOS specification.

Basic component specification A basic component is represented in the LOTOS specification by a process declaration, where the component's ports are declared as the process gates. The body of this LOTOS process is an abstraction of what we know about the component's implementation.

Complex component specification The specification of a complex component is basically the specification of the communications between sub-components. The communications used in COOPSCAN are either synchronous communications between single components, where the sender knows the receiver and is blocked until the communication completion, or one to n communications involving a collection. We make a

```
process CoopScanControl[Join, AddNewConnection,AckJoin,JoinAcknowledged,
              GiveFloor,FloorTaken,AskFloor,FloorDemand,
              UiopSend,UIopGet](Group : SET) : noexit :=
     ( (* Diffusion of a service call *)
     UIopSend ? Site : SiteId !BEGIN;
     Broadcast[UIopGet](Site,Group)>>
     (UIopSend ! Site ! END;
     CoopScanControl[...](Group)
     ) )
□
     ( (* Notification *)
     Join ? Site : SiteId[Site notIn Group];
     Broadcast[AddNewConnection](Site,Group) >>
     CoopScanControl[...](Add(Site,Group))
     )
□
     (
     (*other communications *)
     ) endproc
```

Figure 2 CoopScan controller (partial) LOTOS specification

distinction between a service call, which is a synchronous communication (sender is blocked until acknowledgment of all receivers) and a notification, which is an asynchronous communication (sender does not wait for any response from the receivers).

Every potential communication is described inside the controller of the complex component. This controller is given as a LOTOS process synchronized with every sub-component for every possible services present in their interfaces.

We present in figure 2 a part of the controller of the component *CoopScan* , where we describe the one to n `UIopSend` [*] service call, where the sender needs an acknowledgment from all receivers before continuation, and the notification `Join`, where the sender is not interested in what receivers may do. Furthermore, the session controller keeps track of the integration of a new user by updating its internal variable Group on the action `AddNewConnection`.

The ... notation in recursive calls indicate the same set of gates in the same order as in the process declaration.

Complete specification The complete specification of the COOPSCAN framework is about 3000 lines of LOTOS long, when we consider basic protocols for registration and access control.

3 VERIFICATION

Writing a formal description of this application is in itself a gain, as it allows to unambiguously expose the architecture and choices made for the implementation.

[*] UIop is an operation on shared data, generated by the floor holder and broadcasted to other sites

However, one can go a step further, by using verification tools to analyze and attempt to validate some basic and more advanced requirements. For this work, we used the CÆSAR-ALDÉBARAN toolbox [FGM$^+$92] which integrates a LOTOS compiler and tools for deadlock detection, simulation, temporal logic checking and behavioral specifications checking.

3.1 Verification requirements

In the following, we will give only some of the most significant properties, and describe how we verify them.

Deadlock freedom Deadlock freedom is a crucial property, and one can spend 90% of the verification/correction cycle just to obtain it (or at least understand why and how a deadlock occurs).

In our analysis process, we usually alternate between two main tools of the OPEN-CÆSAR toolbox for deadlock states search : the Terminator tool, which implements Holzmann bit-hashing technique [Hol89] for efficient (but partial) model exploration. When a deadlock is found, the Terminator tool can produce access sequences from the initial state; these sequences can be replayed by the Simulator tool in term of the original LOTOS program actions. The second tool we use for deadlock search is based on a BDD representation of the model. It allows the complete exploration of the model, but does not provide access sequences playable by the Simulator tool. We use it essentially as the final step, to prove the deadlock freedom when Terminator can not find any more deadlocks (or if deadlocks are beyond its reach).

Consistency Consistency concerns two issues : all users have the same view of the shared space (consistency of cooperation space), all sites have the same informations on the session status (consistency of organization space). In this phase of our work, we don't want to consider a detailed description of the cooperation space, which is clearly application dependent. Thus we only attempt to validate the consistency of the organization space, that is the correctness of the control protocols.

Registration protocol We will consider one specific operation : the Join operation. The definition of this operation ensures that the consistency is preserved (any modification of the organization space is broadcasted to all participants), provided that there is no communications failure and that the Join operation succeeds. We have taken the non failure of communications as an hypothesis, so we only have to verify the property that *a join demand eventually succeeds*. This is a liveness property. It is simply expressed by the following formula, expressed in the LTAC temporal logic [QS83]:

$$JoinSession!S \rightarrow Inev(JoinAcknowledged!S)$$

which means that when a *JoinSession* action has been issued by the site S, then any following execution sequences will contain an occurrence of the *JoinAcknowledged* action for the same site. The same kind of verification is done for the QuitSession operation and also for the access control operations, but it will not be described here.

Concurrent Access control In CoopScan the floor control is given to only one participant, usually the session creator. Then there can be different floor passing policy, either based on negotiations or on a fixed protocol. Whatever is this protocol, the following property must hold: there is always (at least) one floor holder.

This is expressed by the more general property that it is always possible to edit the cooperative space, i.e. there always exists a site S which can issue a `UIopSend !S` command. It corresponds to the following formula:

$$init \Rightarrow All(Pot(Enable(\texttt{OpSend})))$$

which means that from all states, there exists an execution sequence containing the action `OpSend`. To verify this property, we simply rename all `UIopSend !S` actions for any S into the `OpSend` action and then we hide all actions different of `OpSend`. We minimize the resulting model with respect to the branching bisimulation. We must obtain a LTS reduced to one state with a `OpSend` loop transition.

Other properties of interest are related to the uniqueness of the token.

4 CONCLUSION

We have presented the formal specification in Lotos of a framework for the development of groupware applications. This formal specification was derived from a component based description ; these components were themselves defined from the actual implementation of CoopScan. We have given a set of generic properties for this kind of framework and verified some of them using the Cæsar-Aldébaran toolbox. Given this verification basis, we can now integrate and test various control protocols, and also test a parallel version of some parts of these protocols. It is especially the case of the registration protocol, which is actually completely sequential: every new comer is treated one after another, by a unique registration manager ; we would like to introduce several registration managers and decide how we can decompose the registration process into several concurrent transactions.

The verification phase of the CoopScan description is not a trivial process, as the model generated is large (over 20 million states at the beginning of the verification process). We have used with significant success the “on the fly” and Bdd based extensions of the Cæsar-Aldébaran toolbox, combined with compositional model generation.

The CoopScan work is used as a case study for a more general project, whose main aim is to develop the components approach for the conception and management of cooperative applications. Our aim in this project is to introduce in a component's description behavioral specifications as an abstraction of its dynamic behaviour. This integration can be made at two levels: in its interface, where we will give a description of the component's intended behaviour, with respect to the services it offers and requires ; in its implementation (if it is a complex component), by a specification of its controller.

The introduction of these behavioral specifications will help to enforce an important need, which is the ability to reuse software components, or to replace a software component by another without interferences. This is actually a growing trend, as can be seen in [AG94][Sha94][YS93], where the authors focus on the notion of connectors, and how one can formally describe them.

Thanks We want to thank Roland Balter for his careful reading and useful comments on this work.

REFERENCES

[AG94] R. Allen and D. Garlan. Formal connectors. Technical report, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, 1994.

[BAK95] R. Balter, Slim Ben Atallah, and Rushed Kanawati. Architecture for synchronous groupware application development. In *HCI*, Tokyo, Japan, july 1995.

[BB88] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. 14(1):25–29, January 1988.

[BBA+95] R. Balter, L. Bellissard, S. Ben Attallah, F. Boyer, D. Demitrescu, R. Kanawati, A. Kerbrat, E. Lenormand, D. Lutoff, V. Marangozov, M. Riveill, and J.Y. Vion-Dury. Objectifs et orientations de l'action olan. Technical report, Bull-IMAG/Systèmes, 2 avenue de Vignates, 38610, Gières, 1995.

[Ca90] T. Crowley and al. Mmconf: An infrastructure for building shared multimedia applications. In *CSCW;90*, page 329, Los Angeles, October 1990.

[Dul94] N. Dulay. Darwin overview. Technical report, DSE Imperial College, London, 1994.

[FGM+92] J.C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodríguez, and J. Sifakis. A toolbox for the verification of lotos programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, pages 246–259, New-York, May 1992. ACM.

[Gro93] Object Management Group. The common object request broker: Architecture and specification. Technical report, December 1993.

[Hol89] G.J. Holzmann. Algorithms for automated protocol validation. In Joseph Sifakis, editor, *Proceedings of the 1st International Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, June 1989.

[MR92] S. Greenberg M. Roseman. Groupkit : A groupware toolkit for building real-time conferencing applications. In *CSCW'92*, page 43, November 1992.

[QS83] Jean-Pierre Queille and Joseph Sifakis. Fairness and related properties in transition systems — a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.

[Sha94] Mary Shaw. Procedure calls are the assembly language of software interconnection. Technical report, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, 1994.

[YS93] Daniel M. Yellin and Robert E. Strom. Interface, protocols and the semi-automatic contruction of software adaptors. Technical report, IBM T.J. Watson Research Center, 1993.