Agent Protocols for Integration and Cooperation in a Design Application Framework

Aiko Frank
Universität Stuttgart, IPVR, Breitwiesenstr. 20-22D, 70565 Stuttgart, Germany,
Aiko.Frank@informatik.uni-stuttgart.de

Abstract

Design is a discipline, which requires support for the combination of different systems and tools, spontaneous interaction between designers, sharing of (design) data, and adaptability to altering conditions. When looking at design processes it is a demanding task to coordinate and organize the work of the different team members. The work of a single person not only depends upon the work of others, but can also influence the work of other designers as well. Correspondingly, patterns of cooperation are to be established to minimize or resolve resulting conflicts, support the concurrent use of data, and to coordinate the work process.
In order to fulfill those requirements the ASCEND Designflow Model and its prototype CASSY offer a corresponding object model, an architecture combining workflow and groupware systems, and finally a protocol integration layer based on agent communication. This protocol layer helps to flexibly combine the functionality of the different systems and services, since it can be adapted to different application scenarios. In this paper, we will describe the concepts of the protocols and their implementation architecture, resulting in a generic protocol engine.

1. Introduction

In this section we describe the motivation for our approach and how protocols help to adapt to different application scenarios as well as integrate data sharing, workflow, and groupware functionality.

1.1. Motivation

Divide and conquer is the underlying technique that allows to cope with the inherent complexity of engineering design. This strategy refers to a decomposition of the *artifact under design* into separate *design items* and to the delegation of *design tasks* to separate *design teams*, with each design team being responsible for the design of the assigned design item. To be able to concurrently work within a team on those partitions without interfering or avoiding to interfere with other designers is a demanding task. Participants in design have to cooperate often by other means than their current design environment; they have to resort to personal communication methods (like e-mail or telephone) or have to follow certain external design guidelines in order to coordinate their work and to resolve conflicts. Since these rules and design guidelines are specified outside the system, integrity cannot be controlled automatically, leading to manual and error prone design control.

It is known that both workflow and groupware technology can be successfully applied in order to support design activities that are carried out by applications, like e.g. mechanical engineering, software design, or planning [10]. However, there is hardly any workflow or groupware system exploited for an entire design process. One major reason for this lack of acceptance is the restrictive applicability of both technologies: workflow technology concentrates on mostly rigid and predefined activity modeling and thus restricts the creativity needed within the design process; a single workflow defined gets instantiated and enacted many times and is processed more or less automatically. Quite contrary, groupware supports cooperation among persons, focussing on powerful cooperation protocols (e.g., data sharing, conferences, group editing); each cooperation being performed differently.

1.2. Related Work

Current CAD or CASE tools, often suffer from a lack of coordination among the ongoing design activities. Most rely simply on version management and shared files systems. CAD framework research, as found in CONCORD [10], NELSIS [13], and Odyssey [3], is more advanced since they provide the coordination of design steps, called *designflow management*. CONCORD and NELSIS also realize some kind of repository for managing design data (i.e., the meta data as well). Odyssey uses a resource manager for data handling.

Still these approaches focus mainly on the concepts of tool integration as well as activity management. Cooperation is often neglected. CONCORD stands out by offering so called cooperative designflows and corresponding coop-

eration operations based on a comprehensive transaction model [10]. But this approach is restricted to a shared data area between cooperating activities. Also operations on the shared data are limited. Basically the available operations are *read*, *write*, *transmit*, *permit*, *revoke* and a notification mechanism. This is a profound base for executing cooperation on data [11]. Unfortunately it is hard to realize specific access protocols for different data types. The CONCORD approach is based on database objects with the corresponding data operations, whereas we prefer to also support higher level operations for the objects (e.g., rotation for CAD objects). At last we want to support direct negotiation between designers, especially regarding the goals of their corresponding assignments. Hence, we believe it to be important to offer cooperation protocols which can be adapted to various application scenarios. Additionally our approach focusses on integrating different systems (workflow systems, groupware applications, data storage, and tools). To realize our model we propose the use of adaptable protocols to deliver the according interactions.

1.3. System overview

By means of the ASCEND Designflow Model (ADM), one is able to specify designflows adapted to an application's particular design process and design methodology. A designflow can comprise complex design activities that are built out of workflow activities and groupware activities. Furthermore the shared data is organized in a shared information space (SIS), as described in [7]. The basic pillars of this model and our prototype implementation CASSY (*Cooperative Activity Support SYstem*) are the common activity management, an object model, a shared information space and the exploitation of flexible protocols to ensure adaptability to different design scenarios. Those protocols act as the glue between the different aspects of the ADM. They can realize cooperation patterns in different ways, ranging from direct negotiations between designers up to handling concurrent access to CASSY objects. CASSY uses CORBA as middleware for integrating the different services and systems. We implemented a workflow facility [4] by wrapping IBM's Workflow System FlowMark to support automated processes. On the other hand we implemented a groupware facility by ourselves for generic cooperative functionality.

There exist three main challenges: activity management, data management, and last but not least the integration and adaptation of these aspects by protocols.

A common activity management coordinates the design steps, called activities. The sharing of data has been realized by introducing an object model for structured data. Below in Figure 1 the basic building blocks of the ADM are presented. Activities represent (complex) design tasks. We allow the hierarchical structuring (*consists of*) of activities, as it is well known from workflow management. The objects are data in the SIS which are used by the activities. They can also be structured by introducing adaptable dependencies like *part of* and *is a* [7]. Further functionality can be exploited by the notion of delegation and protocols describing interactions (e.g. *usage* and *negotiation*) in a designflow. [7] and [6] give a more detailed coverage.

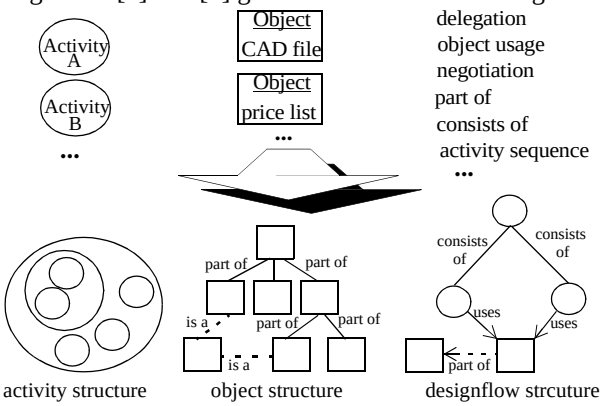

Figure 1. Building blocks of the ADM

One challenge still needs to be discussed: how the interaction between the activity model, the SIS, and the human designers is realized. We propose to use flexible protocols, as presented in Figure 2 to combine the those aspects. We offer access layers to wrap the corresponding systems. This enables us to map and adjust their functionality for our implementation. Groupware and workflow activities are encapsulated by wrappers with interfaces for protocol interaction. Similar the objects for the SIS have to be integrated through the proprietary system interfaces (i.e., we do not store them redundantly). The drawback is to find a compromise between the available system functionality and the desired one. For example, workflow data is not existent if the corresponding workflow isn't running. Also accessing this data is very restrictive, since most workflow systems don't propagate internal states on a fine granular level. Thus we have to take such restrictions into account, by accordingly adding and removing workflow data from and to the SIS.

The ADM itself consists of the system components activity management, SIS, and user interface (Figure 2). They realize the combination of the wrapped functionality. Finally, the integration by protocols gives us the opportunity to dynamically combine the components and correspondingly adapt their interaction behavior. Thus an implementation can be customized to various design methodologies and scenarios.

In the following section we present, what kind of proto-

cols we need and how they are defined. Section 3 describes the means to realize such protocols, whereas in Section 4 a prototype is presented. Finally we close by summarizing our work and suggesting future research areas.

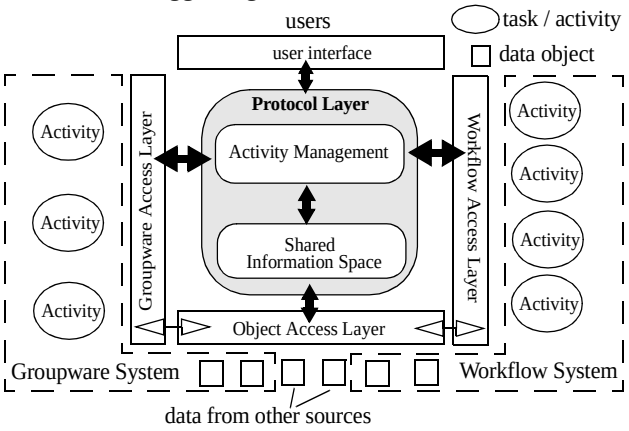

Figure 2. Protocols as integration enabler

2. Explicit specification of protocols

In this chapter we first define the components of protocols. Thereafter some examples will be given to convey the power of such protocols in a design environment.

2.1. Building protocols

A protocol consists of the "control flow", i.e. some kind of state transition graph, the actions caused, and the underlying infrastructure, like messages. A *conversation* is the execution of an interaction between more than one participant based on a common protocol.

Protocol graphs

We use two different types of states. The first one is called *observer* state and depends on an external message to arrive. The second kind is an *internal* state, which is reached by executing a transition through the previous state. Two conversing parties, use couples of internal and observer states to exchange messages, respectively.

Basic message structure

The protocol layer acts as the communication enabler between the entities of the ADM. The Communication itself relies on messages passed between the parties of a protocol. A message contains the following parameters:

- *Topic*: The topic is a short description of the intention of the message, like *write*, *use*, *accept*.
- *Sender*: A message identifies the sender of itself.
- *Recipient(s)*: Also the recipients are specified.
- *Context ID*: This ID binds a message to a conversation.
- *Further parameters*: Those specify additional information (e.g., the new value for a data object, a textual description, references to entities).

Composition of a protocol

A protocol is described in a state transition graph. A state implements the *actions* taken, when the state is reached. Afterwards it checks the post-conditions and decides the next transition(s) to be taken. Start and final states are specifically marked as such.

A protocol for a conversation between two parties consists of an *initiator* protocol and the *counterpart* protocol. Those complement each other (by observer and internal states) and form the *conversation protocol*. If more than two parties are part of a conversation, one can distinguish between either multiple protocol couples or differing counterpart protocols specific to each participant.

For simplicity, we will only describe in the following sections either the initiator or counterpart protocol, since most of the time their structure is identical and they differ only by the transitions (send/receive). If important differences occur, we will specifically mention those.

2.2. Protocol samples

Generally all entities of the ADM can take part in protocols. Also human users should be included by offering corresponding interfaces via some kind of interaction agent. This way, protocol communication can be completely system supported to ensure proper and consistent handling of any entity of the ADM.

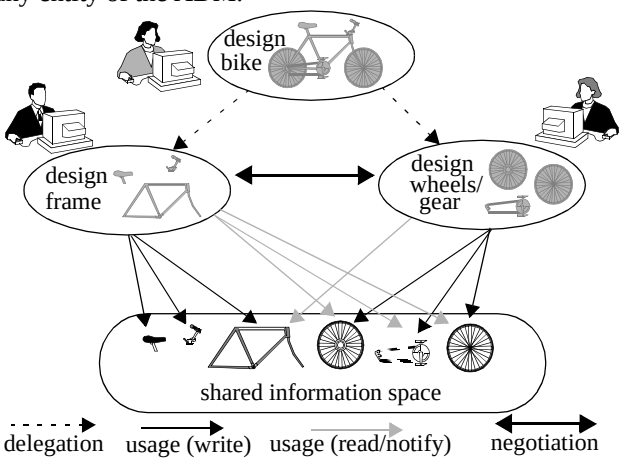

Figure 3. Design process: bicycle

A design scenario

To better understand the application of the following protocols, we describe a simple scenario. The goal of the design process in Figure 3 is to produce the design of a

bicycle. The complete design artifact *bike* is divided into two design item sets (i.e., *frame* and *wheels/gear*). Those are delegated to two different activities, which declare a usage relationship to their respective items from the SIS. Since the wheels and gear depend from the frame design and vice versa, additional usage relationships are established, to be notified on changes. As hinted through the double-arrow the two activities negotiate, when they want to resolve conflicts between their designs.

Negotiation

Negotiation is a generic protocol, which can be used in most situations since it realizes a standard interaction behavior (Figure 4). It starts by a proposal, which is negotiated. This request can be altered by proposing different statements from any negotiation partner and finally be rejected or accepted. The request states the topic and some representation of the content (e.g. "change gear design" or some formal specification). The state diagram shows the requester's view in such a negotiation:

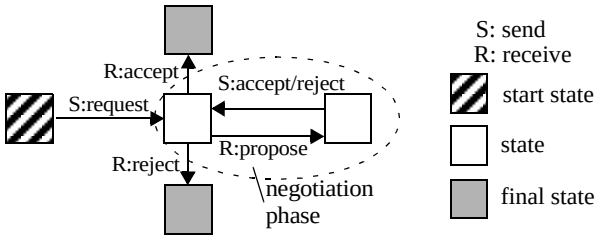

Figure 4. Generic negotiation protocol

This protocol can be exploited in many areas, when no specific protocol exists, but the participants have to converse in an organized way. The genericity is an advantage, since it can be exploited when implementing new application scenarios. Easily a specific negotiation protocol can be instantiated to adjust to the needs of the specific scenario, without implementing the control logic of a conversation, but only by using corresponding topics and parameters.

Usage

An important scenario for the ADM is the concurrent use of shared data in the SIS. For an activity to use an object of the SIS, it first has to register on this object to establish a usage relationship. After a successful registration the activity can issue operation requests to this object (e.g., read, write, or change of access permission). Finally the usage relationship is terminated. It isn't necessary to realize all of the three phases in a single protocol, in contrary it seems to be more efficient to offer different protocols. That allows us to dynamically use the protocol which best fits the current situation and the specific object type.

A usage request is detailed in Figure 5. It differs only slightly from the generic negotiation. But one should note that the topic (*usage*) is predefined as the negotiation phase is also more exactly tailored to the registration (e.g., whether a read lock can be acquired). The termination of an usage is not detailed, as it works as one can expect.

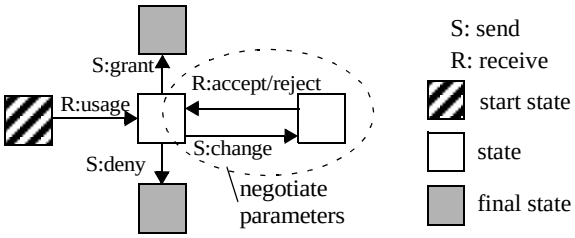

Figure 5. Usage request protocol

Object operations

The operational requests to an object depend on the object type. For example, a database table offers SQL queries and locking, a CAD object might allow its rotation. We look a three examples for a write operation, to see how this can be realized:

- Exploitation of locking mechanism: Only if the writer possesses exclusive write access, the request is executed at once. If write access is not given, the requester can be informed of the owner(s) of write locks (through the parameters of the reject message). Then it lies upon the requesting activity to start negotiating with the owner(s) about transferring the write lock. After acquiring the lock it can try again the write operation.

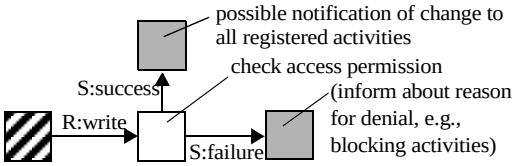

Figure 6. Write operation with locks

- The object is shared by many activities which have to agree on the proposed write (see also [8]). This requires the protocol to find a decision if the request is accepted or rejected (Figure 7). This might seem complicated, but raises consistent views between the activities on the data. Also the notion of group awareness is reached, since the other activities become aware of the changes.
- The writer has no write access, but the object tries on a best effort basis to execute the operation. So the object itself starts negotiating with the owners of write locks to either permit the write operation or to transfer the access rights. If successful, the write operation can be executed. In this case the protocol of the object invokes some other negotiation protocol(s), while it is in the 'check state' of Figure 6.

The basic write protocol remains the same for all presented scenarios, but the implementation of the actions of the states varies. It strongly depends on the intended use of the object. We will discuss this further, regarding customizability in Section 3.2.

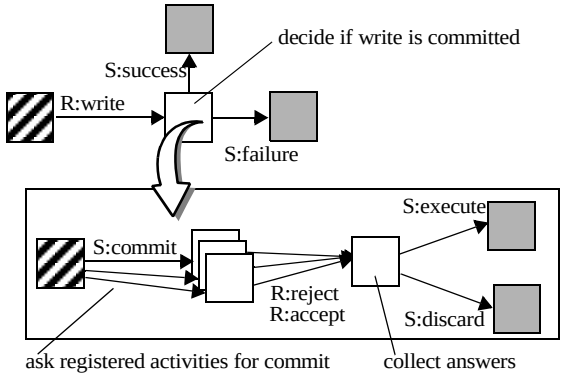

Figure 7. Write operation with multi user commit

Delegation

Design typically relies on decomposing a design artefact into smaller design items and delegating those to different design activities. The *specification* of a design item is the means to ensure a better integrability when combining the single items again to form the desired artifact. As it is well known, often problems arise when designing separate items which have inter-dependencies. Dynamic change of the specification is necessary to adapt to changing situations. Also the integration of the design items can cause problems, which have to be solved by revising design items, thus delegating them again to be worked over.

The delegation protocol supports this process and thus is generally a protocol between activities. It comprises the assignment of a piece of work (i.e., a design item), negotiation about the specification for the assigned item, and the decision whether the task has been completed. The completion does not only depend on fulfilling the specification of a single design item, but might also be deferred until all design items have been successfully integrated.

The protocol for initiating the delegation (Figure 8) transmits the specification and the reference to the corresponding object in the SIS by parameters of the message. The specification can be negotiated (*refine*). A change can be proposed from either activity until the delegation is accepted or rejected.

A later change of specification can be initiated from either side of a delegation relationship. Thus the case of design problems (i.e., the specification can't be fulfilled as planned) and the case of integration problems (i.e., specification was fulfilled but a conflict occurred in the integration) can be handled by our system. Basically this protocol is identical to the initiation, except for the topic *delegate* changes to *modify_spec* (not depicted).

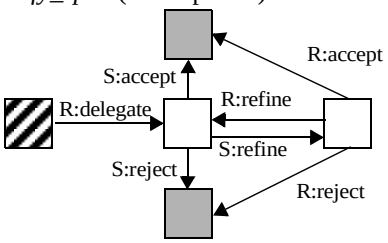

Figure 8. Initiating a delegation

Finally the delegation can be concluded as shown in Figure 9. Here the delegated activity returns with the finished design item. The delegator can decide to accept it or to demand further work (*redelegate*), which has to be specified and can be negotiated (*refine*).

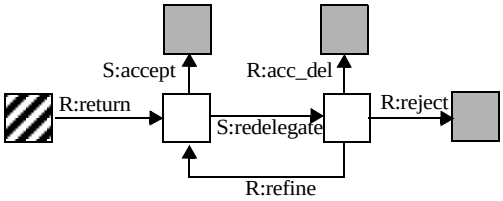

Figure 9. Conclusion of the delegation

2.3. Further areas of interests

As can be seen by the discussed scenarios, it can be necessary to nest conversations, if a request requires involvement from other parties. Thus a state of a protocol can contain the execution of a further protocol. The integration of users into conversation can often be mandatory (e.g., when access permissions should be transferred). So it is important integrate users into the protocol handling.

3. Realization

We currently don't plan to introduce dynamic conversation, as is found in autonomous agent systems. We prefer to pre-define certain interaction patterns, which can be customized to different requirements. In this section we cover the basic ways of customization and introduce our notion of protocol engine.

3.1. Motivation

Our model is very powerful by integrating different technologies. For example, the aspects of activities, SIS, delegation, and conversation combined, allow to control the coordination of work on different levels (Figure 10). For one, work can be partitioned by delegation, where the specification can be dynamically altered in order to prevent and resolve conflicts. Furthermore object access is coordi-

nated by using according protocols. Hence, indirect conflict resolution is achieved. Finally, conflicts are directly negotiated between activities, if all other means fail.

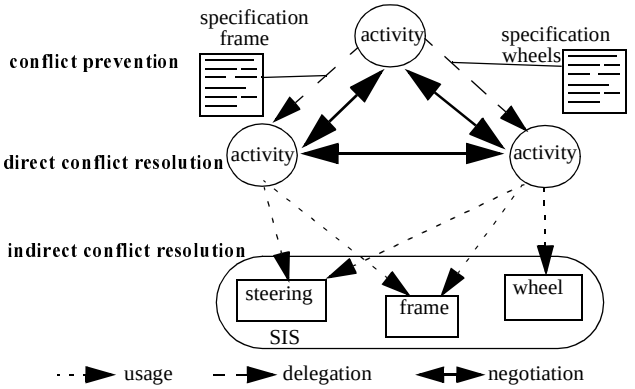

Figure 10. Levels of protocol application

3.2. Customization

Protocols allow to customize the ADM, since interactions can be defined as required by a specific application. Thus an implementation can be adjusted to many different relationships and cooperation patterns. As an example, the ADM supports the introduction of different types of objects. Their usage can be customized on three levels:

- Introduction of a new protocol.
- Use of an already existing protocol and altering it according to the new handling, this can mean for example introducing new topics, parameters, or states.
- Use of already existing protocols and changing only the states' action (e.g., by implementing a different 'check state' in the write example from Section 2.2). This allows other entities to use this object type even without changing their counterpart protocol.

Thus we realize a compromise between dynamic agent conversation and purely pre-defined interaction. Dynamic composition of pre-defined protocols is achieved in our model by nesting. Existing protocols can be re-used through a reservoir of protocols and protocol fragments, which are composed as needed for a specific application (see Figure 11). E.g., a two phase commit protocol (2PC) can be applied in many protocols, where a decision with veto right has to be supported (see Figure 11 ④).

The state implementations show how differently a protocol state can be implemented, without affecting transitions (numbered ① to ③). Nesting is the execution call to a protocol from the protocol reservoir (Figure 11 ⑤). When picking from the protocol reservoir, additional entities can be included into the conversation (e.g., by calling the 2PC protocol on a write request). Therefore very complex conversations can be established.

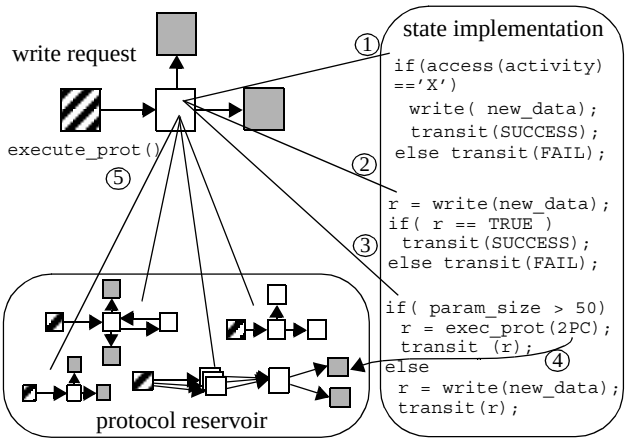

Figure 11. Implementations for write protocol

A further example for such a complex protocol is given in [7], where a (sub) object y is installed in a part-of-dependency with a super object x. This super object has to be queried for a permit for modifications on its sub-objects.

3.3. Protocol engine

To actually being able to exploit the described approach, an implementation has to offer the re-use of protocols. It is obvious that the definition of messages and the nesting of protocols must be also supported accordingly. Finally the integration of the protocol execution into the entities should be of no great effort. Thus we decided on a protocol engine component, which executes arbitrary protocols.

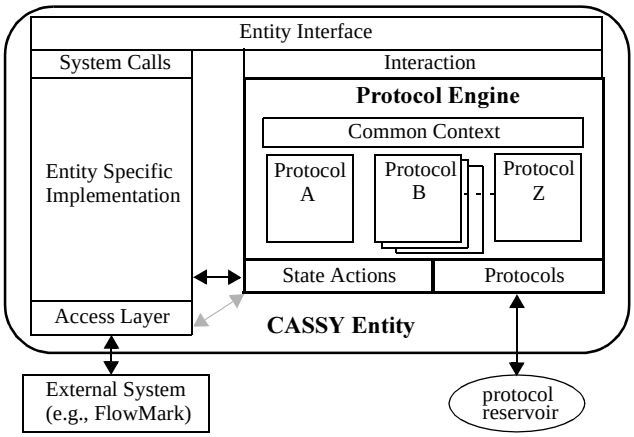

Figure 12. Protocol engine for CASSY entities

This engine is a plug in component for ADM entities. Figure 12 shows the embedding of the protocol engine in a CASSY entity. Calls to the entity are divided in system calls, like registering an object to the SIS, and active interaction between different entities and system components through protocols (e.g., delegation, usage). The protocols are taken from the protocol reservoir and customized by

correspondingly implemented actions. These state actions interact with the specific entity's implementation, hence realizing the object behavior. The entity's implementation relies on the access layers, which are depicted in Figure 2.

The protocol engine has to satisfy the previously stated requirements. The message structure should be defined through some language, which is easy to understand, simple to use, but still powerful enough to describe complex facts. We will discuss this in the next chapter.

4. Implementation

To support the desired protocol engine with its flexible protocols, we didn't have to look very hard. CASSY is based on CORBA objects, which already show some active object behavior. Hence, we found the approach of agent communication promising, especially since it is very adaptable and supports arbitrary interactions. For our prototype implementation we used the Java based JATLite package [9] as platform for agent communication. The message format of our implementation has been defined in KQML [5], because of its ease of use and flexibility.

In our current implementation protocols are constructed in Java classes. An excerpt from the usage protocol class (`class Usage`) is listed below. It extends the class `InteractionDefinition` and implements the initiator part of the usage protocol:

```
public class Usage extends
abcassy.interaction.InteractionDefinition{
public InteractionMachine getInitiatorMachine(
String initiator, String counterpart, String
contextId) {
InteractionMachine rval = null;
ActionRepertoire actionRepertoire =
this.getActionRepertoire();
String sender = initiator; String receiver =
counterpart; String context = contextId;
KQMLmessage USAGE = new KQMLmessage("(use
:object_ID <ID> :usage-type <type> :context " +
context + " :sender " + sender + " :receiver " +
receiver + ")");
// further messages...
ObserverState o0 = new ObserverState("O0");
AbstractState s0=AbstractState.getInstance("S0");
// further states...
MessageObserverTransition t0 = new
MessageObserverTransition( o0, s0, "S(USAGE)",
new Fact(USAGE), null,
actionRepertoire.getSendAction());
// further transitions...
rval = new InteractionMachine(o0);
rval.addState(s0);
//...
rval.addTransition(t0);
//...
return rval;
} /*getInitiatorMachine()*/} //class Usage
```

The listing shows the definition of messages using KQML (i.e., class `KQMLMessage`; the definition of the messages and the implemented protocols are detailed in [2]). Various protocol states are initialized, where a `ObeserverState` is waiting for a messages and `AbstractState` sends messages. Finally transitions between these states are defined. Since Java supports dynamic class loading, such protocols can be loaded at runtime. This directly supports our notion of protocol reservoir, where the protocols can be used as needed, without implementing them anew.

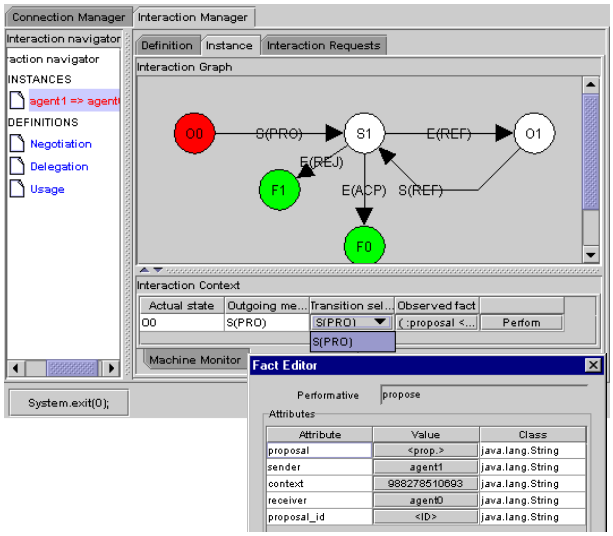

Figure 13. Screenshot: protocol execution

To demonstrate communication we shortly describe an execution of the negotiation protocol, which is depicted as state graph in Figure 13. This shows a client after the initiation of the protocol and the fact editor, which allows to edit the message to be sent. States are dark grey for active, white for intermediate, and light grey for final. Transitions are labeled by S and E for send and receive, respectively. The parentheses contain the abbreviations for the topics (PRO: propose, ACP: accept, REJ: reject, REF: refine).

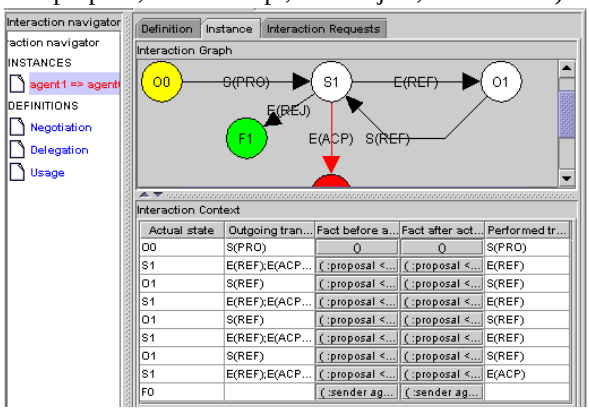

Figure 14. Screenshot: conversation history

The situation is before activating the S(PRO) transition, which starts the negotiation phase. After sending a message, the counterpart client changes the state accordingly and offers the user valid replies. A protocol of a full conversation is listed in Figure 14. As can be seen, multiple refinement messages (S(REF), E(REF)) have been exchanged to modify the initial proposal until an agreement was reached (E(ACP)).

An advantage of this client is that human users can be integrated into protocols. The implementation can offer a user choices of answers (or requests) valid in a protocol, as presented in the list of transitions in Figure 13. Thus the proposed approach allows direct interaction between users and/or entities. We are able to realize protocols between entities, which are automated as far as possible. But when a point is reached where human interaction is required a corresponding choice of messages can be presented to the users. Through this we can integrate the them more tightly into the process, while still relying on system support. Hence, less coordination and cooperation has to be executed outside the system.

5. Summary and outlook

Our approach integrates existing systems and tools, (cooperative) data sharing, and flexible cooperations patterns by exploiting a protocol engine. Hence, an implementation of the ADM can be adapted to many application scenarios by customization, like object modelling and protocol definitions. Even at runtime, protocols can be dynamically invoked to confront newly arising situations. We don't go the way of autonomous agent systems in computing each interaction on demand, instead we do rely on predefined protocols, which roughly corresponds to the concept of conversation plans (e.g., [1]). We use a protocol reservoir to build interaction patterns as needed without implementing agent "intelligence". For sure, approaches which implement complex systems by using only agent technology (e.g., like some workflow management systems) are very interesting. But our aim was to build a framework which can easily integrate various legacy systems and can be applied to many application areas.

Because of the positive results from the current prototype we continue to implement a more sophisticated version. JATLite might not be fully sufficient for CASSY, since we want to more easily support the nesting and reuse of protocols. Also our implementation didn't support the conversation between more than two participants in a single protocol. For the next version an even more intuitive GUI is to be realized to better support integration of human users. Finally we're thinking of using some higher degree of customization regarding the state implementation by applying ECA (event condition action) constructs.

6. Acknowledgements

This project is partly funded by DFG grant Mi 311. Further thanks go to Kerstin Schneider and the reviewers for their helpful suggestions.

7. References

[1] Barbuceanu, M., Lo, W.-K.: Conversation Oriented Programming in COOL: Current State and Future Directions. Workshop on Specifying and Implmenting Conversation Policies of the Autonomous Agents '99, Seattle, Washington, May, 1999

[2] Bozak, E.: Integration of Autonomous Agents in to the Designflow System CASSY; semestral thesis (in German), Univ. of Stuttgart, Studienarbeit Nr. 1769, 2000

[3] Brockman, J.B., Cobourn J.B., Jacome M.F., Director S.W.: The Odyssey CAD Framework; IEEE DATC Newsletter on Design Automation, 1992

[4] CoCreate Software et al.: Workflow Management Facility, 1997 OMG Document: bom/97-08-05, 1997

[5] Finin, T., et al.: KQML: A Language and Protocol for Knowledge and Information Exchange. In: Fuchi, K., Yokoi, T.: Knowledge Building and Knowledge Sharing. Ohmsha and IOS Press, 1994

[6] Frank, A.: Towards an Activity Model for Design Applications. ISCA 14th Intern. Conference on Computers and Their Applications, April 7-9, Cancun, Mexico, 1999

[7] Frank A., Mitschang, B.: On Sharing of Objects in Concurrent Design; accepted for the 6th International Conference on CSCW in Design, London, ON, Canada, 2001

[8] Frank, A., Sellentin J., Mitschang, B.: TOGA - A Customizable Service for Data-Centric Collaboration. Information Systems Journal, Vol. 25, No. 2, Elsevier Science Ltd., UK, 2000

[9] Jeon, H., Petrie, C., Cutosky, M. R.: JATLite: A Java Agent Infrastructure with Message Routing. IEEE Internet Computing, IEEE, 2000

[10] Ritter, N., Mitschang, B., Härder, T., Gesmann, M., Schöning, H.: Capturing Design Dynamics - The CONCORD Approach, Proc. 10th Int. IEEE Data Engineering Conf., Houston, US, 1994, pp. 440-451.

[11] Ritter, N.: Group-Authoring in CONCORD - A DB-based Approach; 12th Annual Symposium on Applied Computing (SAC'97), San Jose, U.S.A., 1997

[12] Winogard, T., Flores, F.: Understanding Computers and Cognition: A New Foundation for Design; Ablex, U.S.A., 1986

[13] van der Wolf, P., Bingley, P., Dewilde, P.: On the Architecture of a CAD Framework: The NELSIS Approach; Proc. European Design Automation Conference, pp. 29-33, March 1990