Integrating Object and Agent Worlds

Thomas Rose
Dept. of Computer Science
University of Toronto
Toronto, M5S 1A4
Canada

Carlos Maltzahn
Universität Passau
Innstr. 33
D-8390 Passau
Germany

Matthias Jarke
RWTH Aachen
Ahornstr. 55
D-5100 Aachen
Germany

Abstract: Repositories provide the information system's support to layer software environments. Initially, repository technology has been dominated by object representation issues. Teams are not part of the ball game. In this paper, we propose the concept of sharing processes which supports distribution and sharing of objects and tasks by teams. Sharing processes are formally specified as classes of non-deterministic finite automata connected to each other by deduction rules. They are intended to coordinate object access and communication for task distribution in large development projects. In particular, we show how interactions between both sharings improve object management.

Keywords: software repositories, team support, access and task coordination

1 Introduction

In today's software development environments object repositories play a key role. While early repositories merely provide the service to manage evolving objects, more recent approaches—AD/Cycle, CDDplus/Cohesion and PCTE—deploy repository technology to integrate environments layered around an object management system. Maintaining the consistency of software and software-related objects emerges as a crucial challenge requiring contributions from distinct perspectives.

Each object management system embodies a distinctive *meta model* of software objects and software processes. This meta model, even if not often explicit, describes the structures and mechanisms [18] the object repository is designed to support. Traditionally, environments are tailored towards *information processing* [8, 22]. Software and project databases account for complex structured systems which exist in several versions. Their underlying meta models focus on the products of software processes. Their goal is to capture the ***world of objects***. Configurations and versions are necessary concepts since they provide a pattern to organize systems and components.

Yet, software systems are built by human designers. The increasing size and limited time of software projects place a growing importance on *group processes*. The size and

This work was supported in part by the Deutsche Forschungsgemeinschaft under Grant Ja-445/1-2. Work of the first author is funded by the Federal Networks of Excellence programme through the Institute of Robotics and Intelligent Systems (IRIS).

distribution of development teams result in the necessity to support *collaboration* and *communication* in the ***world of agents***. Collaboration refers to the access and maintenance of objects evolving across a network of agents, and communication refers to the exchange of messages related to the access and change of objects.

What then are reasonable meta models which embody group processes and are feasible to relate group and information processing? In terms of conceptions, one may distinguish connectivity, interoperability and cooperation. *Connectivity* refers to the facility to exchange data physically, possibly across world-wide networks. The ability to exchange semantically meaningful information emerges as *interoperability*. Connectivity and interoperability provide the technological infra-structure for cooperation which is about to enhance individual work by contributing to a common task. Yet, the larger the corporation the more *coordination* of collaboration and communication becomes necessary [8].

For the most part, cooperation support in current environments is fixed to one *protocol*, which is hard-coded into the system. Such technological protocols structure group processes by prescribing patterns of consistent communication and collaboration. But, one distinctive feature of group processes is that groups establish *social protocols* and dynamically adjust them. Thus, fixed technological protocols may be overly constraining. Unstructured communication and collaboration might work well for small groups, but likely cause information overloading and untraceability of development processes in large teams. However, merely focusing on "neat" communication and collaboration patterns will likely result in approaches neglecting the world of objects.

This paper presents the concept of a ***sharing process*** as a model for integrating object and agent worlds. A sharing process describes consistent participation of agents in group processes through classes of non-deterministic finite automata related to each other by deduction rules. Instances of sharing processes govern the distribution of objects across networks of workplaces—called *object sharing*—as well as the delegation of tasks among agents—called *task sharing*. Different instances may adhere to distinct protocols which are formally represented in a knowledge representation language. Since sharing processes are part of our meta model, we are in the position to formally:

- represent distinct object sharings and workspace structures, e.g. optimistic and pessimistic approaches, and
- establish interactions among different instances of sharings, e.g. the admissibility of object modifications with respect to assigned tasks.

This paper is organized as follows. Section 2 draws the development world and reviews proposals to capture the world of objects and agents. Section 3 introduces the concept of a sharing process. Subsequent sections present the formal representation of object and task management and reasonable interaction among them. A prototype implementation *ConceptTalk* is presented in section 7.

2 State-of-the-Art

When people think about humans and group processes in software environments, they think about shared access to repositories, electronic mail communication for notification services, monitoring of task assignments, or collaboration among development processes.

Transaction management is the traditional database technology that provides reliable shared access to repositories. In the *reserve-deposit* paradigm, objects are shared in a serialized way which makes the coordination formally sound but factors out cooperation. Nested transactions just refine this idea [2]. They cluster groups of designers with respect to common, closely-related objects, or vice versa. Making users aware of important changes to their work is outside the scope of the transactional approach. Designer productivity is lowered because of limited parallelism and superficial conflict avoidance [4].

The *copy-merge* paradigm [1] allows parallel changes to one object possibly due to distinct tasks. Parallel changes may cause conflicts which could be resolved by merging. Copying and merging allows closer collaboration, but for the price of additional coordination. The repository should monitor object distribution as well as changes.

In both cases, collaboration can be improved by ***notification services*** triggered at the time objects are changed by, or transferred between, designers. However, notification services must be structured to avoid information overloading. In [13], content and routing of messages are determined by object relationships. Agents are notified about those changes affecting their own modules. But the problem is merely viewed from the angle of object management.

Workspace hierarchies have been introduced to structure group processes. Objects and changes are allocated to hierarchies of workspaces tailored to the architecture of systems [14]. Highly interrelated objects, e.g. due to module relationships, are allocated to sibling workspaces. Changes are merged from sibling to parent workspaces which likely require the most intensive collaboration because of their tight interrelationships. Structures along this line are intended to minimize necessary collaboration and to impose an organizational pattern on group processes. Although these structures are motivated by quality assurance experiences, they impose a fixed structure on communication and collaboration. Hence, system structures pose fixed limitations on group processes, even though there is strong evidence that group structures and processes have a major impact on system architectures [7]. When upgrading to a new operating system version, for instance, task force groups cross fire walls of workspace hierarchies and therefore might face serious problems. Thus, limiting communication and collaboration to workspace structures determined by system architectures is overly constraining. However, workspace structures are a feasible context mechanism to monitor object distribution and evolution, if they could provide more flexibility. We shall refer to this object management within groups as ***object sharing***.

A recent survey concludes that transaction concepts need more *semantics* for further improvement, and that these semantics can only come from information about the *tasks* to be performed by the team [3]. Where does this task information come from ?

Ideas originating from ***distributed problem solving*** [23] and ***office information systems*** emphasize social structures and procedures in teamwork. The outcomes are still the objects, but they are considered side-effects of communication activities. Systems influenced by research in speech act theory manage communicating agents rather than resulting actions and objects. Such systems include discussion tools to figure out design decisions, as in gIBIS [6], and to record the justification of design decisions [19]. Others account for assigning and monitoring contract-based working activities, like Coordinator

[25]. They view agents and their activities as parts of group processes which share tasks to be processed. Conversation structures describe consistent participation of agents in communication processes. This shows a different kind of integration. Agents articulate their contributions at different workplaces. Then, how do these articulations fit together ?

However, systems like gIBIS or Coordinator do not formally relate to the object world. We shall refer to this object-independent management of tasks within and by groups as ***task sharing.***

To conclude, there have been two lines of management approaches, one for *object evolution* and the other for *tasks affecting objects*. The challenge addressed by the ***sharing process approach*** is to integrate both, object management and group process aspects. To capture both worlds, sharing processes have to feature:

- different kinds of agents, like human agents performing tasks or technical agents managing objects,
- different kinds of interaction between agents, like emailing—e.g. where agents are used to communicate to each other—and object tracing—e.g. where technical agents are used for object management, and
- a synchronization facility which allows the specification of consistent patterns of interactions among human and technical agents in a flexible and adoptable kind.

3 Sharing Processes

The sharing process concept allows agents to share objects and tasks within an environment which is comprised of a set of workplaces connected to a network. Formally, a sharing process can be defined as a triple $< E, N, S >$ consisting of events E, a NFA N (non-deterministic finite automaton) and a state S determined by the history of events. Many sharing processes can have the same NFA structure and many events can be associated to a single process.

A *sharing process* documents and structures *events* (fig. 1). Conversely, events drive the evolution of a sharing process. An event is caused by an *agent* at any place in the network. Agents can be human designers who might communicate to other designers to assign a task, as well as technical agents, for instance, workspaces which acquire objects from other workspaces.

NFAs provide synchronization by specifying the consistent evolution of sharing processes in terms of states and transitions. A process has one start state and possibly several final states. Transitions describe state changes and represent the progress of a process with respect to selected final states, the goal states. An event may non-deterministically cause different transitions depending on the state of the sharing process and the agent that delivered the event. In addition, events can trigger notifications to agents.

Conversely, rules can be associated with transitions that trigger consequent events in parallel-running processes of a specified type. In the following, we often use the terms event and transition interchangeably where no confusion can arise; note, however, that transition is a formal concept, used in a limited modeling framework for understanding a wide variety of real-world phenomena, the events. It could well be that events happen that do not really fit the automaton schemata, and we want to be able to record such "exceptions" in our model, hence the separation in fig. 1.

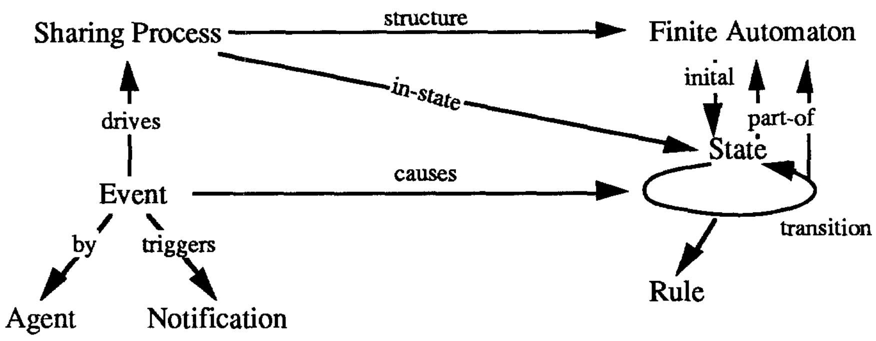


Fig. 1: Sharing processes

Sharing processes are intended to embed object evolution into the more general context of a development project. Each project initially appears like a task and comprises a possibly ordered set of sub-tasks. The state of a project is represented by the tasks completed at a given moment. A project reaches its final state when each sub-task has been completed. The input and results of tasks are managed by *workspaces* which are assigned to groups and single agents respectively. The top-level workspace contains fully integrated "final" project results, whereas lower levels in a workspace hierarchy contain more private and less integrated objects [2, 14]. Due to the conceptual similarity of work organization and work environment, we can associate workspaces with hierarchically organized projects that view objects. Hence, task responsibility assignment and object visibility by certain groups of agents must go together. This does not mean that both have to be identical, but they have to adhere to some kind of well-formed patterns.

In subsequent sections, we show how to utilize sharing processes to *define visibility by object sharing and modifiability by task sharing* in terms of classes of sharing processes. These classes represent schemas for sharing inside a specific environment. Specific objects, specific tasks and changes to specific objects are instances of these classes. Consistency of individual sharings is controlled by instantiation of the classes.

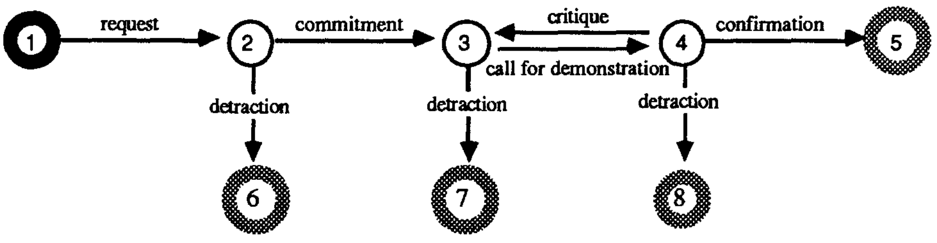
4 Task Sharing

One instance of sharing processes is the sharing of tasks. Task sharing processes handle the execution of tasks which has to adhere to certain protocols. The definition of these protocols depends on the work setting; a particular group may be in different settings at different times or may even adapt standard protocols to their specific needs dynamically. We can abstract from such options by looking at a broader phase structure for task sharing, consisting of the four phases of *orientation*, *assignment*, *realization*, and *inspection*.

During *orientation*, agents identify a problem in one way or another and may come up with the definition of a set of tasks to be executed (cf. [9]). After a task has been identified, it enters the ball game which is about realizing the task. If the task consists of sub-tasks, then the task's sharing process forks into sub-processes, one for each sub-task.

Realization and *inspection* can be tackled within different work structures [4]. An *integrative work structure* involves the whole group; this would be an application domain for real-time collaboration tools that increase group awareness and intensify cooperation [8]. A *delegative work structure* is more formal. It usually separates the realization and inspection phase even though it may iterate through intermediate reviews and similar techniques.

While early workflow protocols were based on the assumption that events would just happen "on command", current conversation-based approaches typically begin with a negotiation phase followed by an asynchronous work mode. The *assignment* could follow an *electronic market protocol* with group members or external agents "bidding" for tasks [23], or a bilateral haggling according to *conversation-for-action protocols* with message types such as request, offer and counter-offer, promise and renege [25].



1: initial	4: inspection phase	7: realization withdrawn
2: orientation phase	5: task completed	8: inspection declined
3: realization phase	6: orientation detracted	

Fig. 2: A phase structure for delegative task sharing

Fig. 2 abstracts from individual protocols by defining an event type called *commitment* which defines the transition from the task definition to the realization phase where the task is worked on in a subordinate workspace. Some time later, a *call for demonstration* initiates an inspection phase in which members of the task sharing process evaluate the results. In practice, these results are often called beta versions; they are distributed to all agents who have influenced the task definition. Usually, errors and weaknesses are detected and lead to events of type *critique* which have the task sharing subprocess return to its realization phase. If, however, the group is satisfied with the results, the subprocess goes into the successful final state by transition of type *confirm*. Further, processes can terminate at any phase unsuccessfully.

To reiterate, fig. 2 shows an abstracted schema for delegative work structures. This schema can be specialized with particular market or bilateral conversational protocols. For example, the ConceptTalk prototype adopts a Coordinator-like protocol [25] for bilateral task sharings but allows users to augment and alter it dynamically.

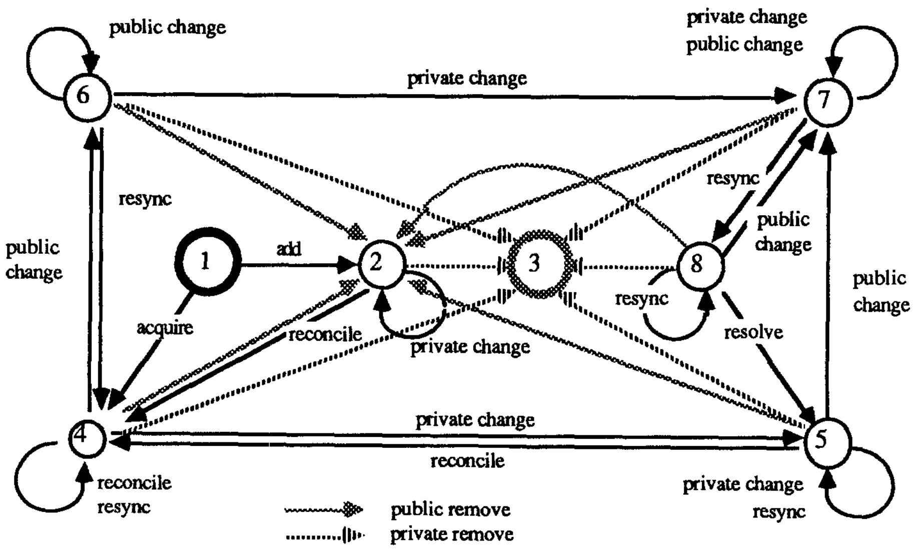
5 Object Sharing

A second instance of sharing processes is the sharing of objects among workspaces in the network. In the sequel, we assume that each workspace constitutes the workplace of a human designer but may be shared with other workplaces. The generalization to the case where each workplace consists of multiple workspaces—the designer's working environment—does not pose major problems in our model but may of course be an interface organization problem [10].

If copies of objects are distributed across workspaces, parallel accesses within different workspaces create multiple variants of an object. Object sharing processes conceive workspaces as technical agents, which manage objects and communicate to coordinate object accesses and changes. To support distributed object management, object sharing processes should feature:

- creating and maintaining *availability* of objects in workspaces, possibly with different access rights,
- maintaining *relationships* among workspaces—e.g., hierarchies—and controlling adherence of availability with respect to these relationships,
- managing *version histories* of objects within and across workspaces, and
- monitoring and controlling *concurrent access* to objects to detect and notify conflicts.

Each object sharing process handles the sharing of one object across two workspaces. Both workspaces contain a copy of the object. A object sharing is initiated between workspaces $\mathtt{work}_1$ and $\mathtt{work}_2$ when $\mathtt{work}_2$ acquires an object from $\mathtt{work}_1$ for the first time. It terminates when the object is removed from $\mathtt{work}_2$ or $\mathtt{work}_1$. The workspace, an object originates from, is called its public workspace. The acquiring workspace is considered a private workspace relative to the public one. The state of an object sharing represents how the private version relates to the public version. The state evolves due to changes to the object in one or both workspaces and transfers of the object among both workspaces. Thus, the process structure specifies concurrent access to shared objects.



1: initial	3: removed privately	5: changed in private	7: parallel modification
2: private copy	4: identical	6: changed in public	8: conflict

Fig. 3: Structure of object sharing for "copying" objects (copy-merge paradigm)

One possible instance of concurrency control specified as a object sharing process shows fig. 3. It adheres to the copy-merge paradigm offered by SUN's Network Software Environment NSE [1]. Such a object sharing protocol supports an integrative work structure and may thus lead to a better overall group result than the usual reserve-deposit paradigm, provided sufficiently dense group collaboration can be achieved [4].

Using the same general approach, one can define different versions of concurrency control depending on the project requirements; in particular, for larger groups, the familiar check-in/check-out locking mechanisms can be used to provide more control (cf. fig. 4 for its fairly simple protocol). But we have to be careful with the concurrent existence of different protocols [24]. This problem is part of the more general issue of how to define the interaction of multiple sharing processes, to be discussed next.

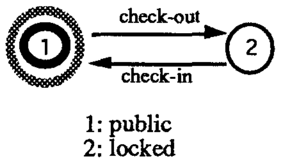


Fig. 4: Structure of object sharing for "moving" objects (reserve-deposit paradigm)

6 Integration of Object and Task Management

So far, we have two types of sharing processes, one to manage object changes and the other to monitor the evolution of tasks. Object sharings realize a transaction mechanism in its own right [1]. Yet object sharings go a step beyond transaction concepts, since they incorporate a novel feature into the nature of a transaction, the *change*.

Transaction concepts process a change in terms of its pre- and post-transaction value. Integrity constraints may place more restrictions on the change. Yet, transactions neither consider the agent who is doing the change, nor possible relationships to other transactions and their intended change. Even the nature of involved objects is not considered. Take a bug-fix in a component of a large software system. Two kinds of information can be utilized: the character of the component and the experience of the programmer. Both have impacts on the change, i.e., the transaction to replace the buggy version by the fixed version. If it is a fringe component, any programmer is qualified to make the change. If it is a central component of the system heavily used by other components, access should be limited to experienced programmers, preferably those having a record of this component. Both kinds of information are not captured by a transaction—i.e. the concept is invariant with respect to objects, callers, and intended modifications—but are relevant to a change in the context of group work.

Object sharing is intended to utilize different sources of information. Some information might originate from a recording of design processes [12, 20] or others from a reverse engineering tool that measures structure [16]. On the other hand, programmer profiles can

[1] [BK 91] surveys the virtues and limitations of transaction concepts originating from both communities, databases and software development environments.

be obtained from recorded task sharings (success rates, in-time ratios, etc.). Some features could merely be designed for describing admissible object sharings, such as "no acquisition of a vulnerable component by an agent with a low profile".

Object sharings and task sharings are also a valuable source for determining the consistency of an object change. *Possible conflicts* arise, when two agents acquire the same object for modifications. They may become an *actual conflict*, when both agents modify and reconcile the object into a common parent workspace. Thus, we have to interface different object sharings.

Furthermore, object sharings have to adhere to some project discipline. It is fine when someone acquires an object to make some quality improvements, but it is bad policy to allow agents to reconcile arbitrary objects and changes into public workspaces. Thus, the eligibility of object changes depends on assigned tasks. This does not imply that only those changes to objects are eligible which have been agreed on; but, the agent who is responsible for the public workspace must be contacted before reconciliation.

6.1 Coordinating Object Changes across Workspaces

A workspace manages objects: objects are available in workspaces and a workspace "stores" the version history of objects. Hence, a workspace appears like a local library which is accessible by owners of the workspace.

An object sharing establishes a communication line between two workspaces $\texttt{work}_1$ and $\texttt{work}_2$ at the time $\texttt{work}_2$ acquires an object from $\texttt{work}_1$ with respect to a specific object `obj`. However, it only defines the availability of `obj` in $\texttt{work}_2$, the private workspace. Availability in $\texttt{work}_1$, the public workspace, is defined in a further object sharing that has the public workspace $\texttt{work}_1$ as its private workspace, and so forth up to some workspace in which the object has the status "only private". Two questions remain:

- how to propagate changes to related workspaces with respect to the object ?
- how to structure relationships among workspaces ?

The first question accounts for the problem of concurrency control among different sharings of the same object. For example, the product's integrity requires that a *public change* transition appears in all object sharings that define availability of the same object with this public workspace. Optimistic policies as in fig. 3 as well as pessimistic policies as indicated in fig. 4 must be coordinated in this manner.

The second question refers to workspace structures—e.g., should a workspace structure be tailored to the system architecture or the group structure. For example, if objects are organized hierarchically in a complex object model, the change of component objects may also cause dependent *private change* events in all workspaces which contain configurations with this component.

Both objectives may appear orthogonal at first sight—at least from a formal point of view. In fact, they are dependent, because of the objects they are governing. For example, a system may be configured from components interrelated by "uses" relationships. Thus, a change interferes with appearances of the same object in other workspaces and it might possibly affect other objects, which are related because of the system structure. Thus, coordination of object changes has to consider multiple objectives. Hence, flexible couplings of object sharings are necessary; yet workspace structures should adhere to some empirical policies [4, 14]. The sharing process approach, especially the rule-based coupling of sharing process protocols, provides a flexible environment in which such policies can be defined and maintained.

6.2 Controlling Object Changes from a Task Perspective

Whenever a subprocess of a task sharing enters its realization phase, an object sharing is initiated that makes the required objects available (by physical copy or by access rights) to the workspace of the server agent of the subprocess. Since the results based on these objects may have to be re-integrated later on, the workspace is subordinated to the workspace where the objects come from. Briefly, the *commitment* in the task sharing causes a *copy* in the object sharing.

Conversely, if an agent attempts to integrate a private object into another workspace in the object sharing protocol, this may have consequences in corresponding task sharing protocols. If the agent is already involved in a commitment for this task, the attempt to *integrate* would cause a *call for demonstration* in the task sharing.

Several kinds of ***conflict*** may occur. ***Differences of opinion*** are recorded in idea sharing protocols, as rationales for design decisions [19]. ***Differences of interest*** can be worked out in the negotiation loops of the task sharing protocols. *Technical conflicts* due to parallel work on the same objects are resolved either formally, under consideration of the task structures, or by introducing an integrative work setting where changes are merged. Thus, our approach accepts the existence of such conflicts and provides ways to make them productive, rather than suppressing them like transactions do. We give two examples that highlight the integration of object and teamwork management.

Creativity in design teams may be enhanced by encouraging unsolicited contributions. When a user *acquires* an object on which another user already works due to a contract, a notification is sent to set up direct communication among the two users. They may then choose to (1) collaborate in an integrative work structure without concurrency control, (2) define notification services which inform one user when the other has completed certain critical subactivities after which parallel work is meaningful, or (3) work in parallel and resolve conflicts by merging results.

The *reconcile* transition of the object sharing protocol triggers two checks. With respect to other object sharings, reconcile may lead to a *conflict* state; in this case, it is aborted and conflict resolution (*resolve*) must precede a new attempt to *reconcile*. With respect to task sharing, it must be evaluated if there is a task sharing between the owners of both workspaces which is in the realization phase for the objects in question. Otherwise, a notification is sent to the originator of the *reconcile* command to start such a task sharing.

6.3 Managing Ideograms Evolving in a Network

To exemplify the different possibilities for couplings, imagine a project and a distribution of objects as shown in fig. 5. Among others, there are three designers—*Mike*, *John* and *Christian*—working on one project developing ideograms. Their private workspaces and the public workspace are connected to a network, a local or an organization-wide one. The public workspace manages entirely configured ideograms as well as parts of ideograms, some of them having revisions and variants. Versions are related by horizontal lines. As of now, there exists one configured version of the ideogram in the public workspace, the "girl" shown at the left-hand side. So far, the setting mimics object management systems.

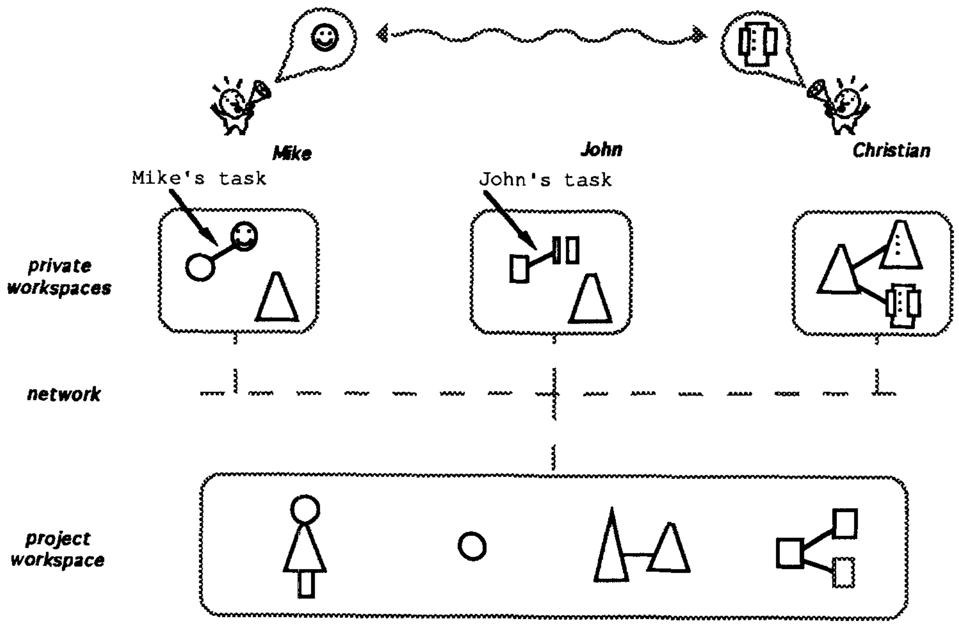


Fig. 5: Ideogram management in a network

Mike's task is to refine the head component to produce a more appealing version of the current ideogram as managed by the workspace of the project. A task sharing between Mike and its client establishes this task. Mike's client is the ideogram manager who administers the project's product. Mike acquires the latest version of the head because of his task, which results in an object sharing between both workspaces. Acquisition might be done automatically at the time Mike has committed to the task—e.g. determined from structured emails—or manually by a designer and his workspace respectively any time after his commitment.

In the meantime, John is working on his task to refine the bottom part. To check compatibility with respect to the central part, John also requires access to the middle part. Mike and John have decided to get a copy of the middle part, since both require a stable context, i.e., changes to that component should not corrupt their work. If a new version comes along in the project workspace, they will receive an email notification and then decide whether to re-synchronize their private version. The latter is again a specific coupling tailored to their needs.

In our setting so far, there are two task sharing and four object sharing processes alive, plus additional object sharings between the public workspace and its client workspace(s) which govern the objects residing in the public workspace.

- The two task sharing processes concern the assignments of the tasks *improve-head* to Mike and *refine-base* to John. The client in both task sharings is the ideogram manager.
- Two object sharing processes are initiated between the public and Mike's workspace, and between the public and John's workspace. They concern the head and the bottom part respectively. Both object sharings start in state *identical* and enter state *changed-in-private* after some modifications have been done in the private workspace (cf. fig. 3). Both changes are associated to a task. The other two object sharings concern the body part. They are still in state *identical*, since nothing happened to the body in both, the private and the public workspace.

After some time, the third designer *Christian* acquires the body part. A third object sharing for the body part starts. In contrast to Mike and John, Christian starts changing the body part significantly yet there is no agreement on this job, i.e., no task sharing has been established. Although Mike acquired the object first, he has not gained the right to prohibit Christian from changing the body. That is a matter of the protocol used. However, Mike and John may want to get a notification. The actual sharings are related by the object, of the project workspace, which they share as common source. Hence, parties to be notified will be derived by deduction rules.

The purpose of their acquisition was a stable context, since they have to interface their parts with the body. Contrary, the acquisition of the head part by Mike is intended to be more exclusive. Suppose, Christian wants to check how his new body versions fit the head and bottom part. When acquiring the head part for instance, he gets a note that this object is in change. Christian can contact Mike to get an intermediate version—Mike may reconcile his current version to the public workspace—or wait until Mike finishes his task and commits the change.

A conflict arises when Mike or John have to adapt their copy of the body part in order to make the body fit to their parts. Then, a conflict occurs and changes have to be merged by Mike and John. Their progress cannot be rolled-back due to a shared componet that requires minor adaptations—like transaction concepts usually do. Further, Christian's change has to be considered too. Christian, Mike and John may communicate to agree on the party responsible for merging their changes.

Since object sharings tolerate conflicts, inconsistency matures as a matter of management, in contrast to the reserve-deposit paradigm. Object sharing protocols manage inconsistency, since they detect conflicts, notify about conflicts and monitor conflict resolution. At first thought, tightly interfacing object and task sharing appears to better designer comfort and enforce stronger policies on object modifications. Interfacing allows the system to provide necessary objects automatically and ensures that no object modification happens without task permission. But, at second thought, a tight and rigid coupling could come in the way of designer productivity and momentum. One may want to develop different conflict notification and resolution protocols. Such a diversity should consider the kind of object and its role inside the system, the kind of change—viewed from an angle of content as well as project deadlines—and the profile of the party responsible for the change.

The examples show just a few of the many possible couplings among sharing processes. Which of these are actually realized, is a management decision. Despite all the flexibility of the sharing process approach, each coupling determines yet another technological protocol that impacts workflow and habits in design teams. The need for formal couplings should therefore be carefully considered in each case. Tools are needed by which the group can re-define the terms of their interaction, without having to go to some systems specialist. Otherwise, we see little chance of acceptance for systems based on these models—or any other models, for that matter.

7 ConceptTalk

Another prerequisite for successful organizational implementation is that the introduction of team support should be unobtrusive. Existing work practices for communication, collaboration, and coordination should be altered as little as possible [21]. The design of the ConceptTalk prototype [15] has tried to follow these premises as far as possible, firstly by using meta-modeling techniques instead of hardcoding of protocols, secondly by building upon standard software environments rather than starting from scratch.

The UNIX world was chosen as an example of a standard development platform. For our experiments, we assume that one is familiar with the following kinds of UNIX tools:

- an object management system (NSE [1]) which offers workspaces as well as simple version and configuration management (based on SCCS and MAKE),
- standard electronic mail, and
- the basic command set of Unix (or a fancier user interface on top of it).

ConceptTalk itself is a small C program that integrates such tools, based on protocol, state, and rule information obtained from a background knowledge server, *ConceptBase* [11]. ConceptBase uses the conceptual modeling language Telos [17] for the formal representation of sharing processes and its instances. Besides the abstraction facilities of object-oriented data models, Telos contributes two important features to our approach: assertional facilities and metaclasses. The assertional facilities come along as integrity constraints and deductive rules. Constraints and rules are heavily used in the coupling of sharing processes. Metaclasses are a natural way to integrate distinct protocol and object models. One instance of a metaclass is the concept of a sharing process which introduces the concept of *agents*. The classes of designers as well as workspaces in the network are represented as instances of agents. In [20], the use of metamodeling for integrating group models, as proposed here, with specific object models is demonstrated.

Communication via *mail* and collaboration via NSE is coordinated by Telos models. As a specific task sharing protocol, we use a slightly modified version of the one proposed in [25]. Since this protocol is not real-time, each of its message types is directly associated with a state transition; the overall structure elaborates the phase model shown in fig. 2 by various negotiation loops. ConceptBase offers a choice of graphical and textual editing tools for changing protocols interactively, even while conversations under these protocols are going on. Flexibility in sharing processes is also provided by the option of moving up and down in generalization (*isA*) hierarchies of existing protocols; the top of such a hierarchy could be the simple *send-respond* pattern of standard email. Finally, the environment also offers a few specialized shared-window tools for real-time collaboration on common workspaces, including a specialized graphical editor and a public domain conferencing software for informal chatting over low-bandwidth channels.

For cases where no graphical interface is available, a simple command interface allows the use of ConceptTalk in standard Unix environments. ***Confer*** is an extended *mail* that offers the available message types at any moment in time and records the exchange of typed messages; the user can review the state of the task sharings in which he is involved as a customer resp. contractor. ***Note*** allows each user to define personalized notification services which "observe" either an individual process or some class of processes with respect to particular events. The events are specified either by a general class of events or by some structure element of the NFA, e.g., an edge; the available classes of processes and the already existing notification services can be shown. Finally, ***pose*** structures workspaces as project processes by associating scheduling information (e.g., problems, tasks, and dependencies among them). This information can be either completely informal (e.g., text, drawings), or represented in Telos.

In line with our philosophy of changing the existing work environment as little as possible, object sharing directly uses the NSE commands *acquire, resync, resolve,* and *reconcile* [1] to denote the corresponding state transitions in the protocol automaton shown in fig. 3. However, in ConceptTalk, these commands to not work autonomously but are controlled in ConceptBase with respect to their feasibility and consequences, especially in terms of coupling with other result and task sharing processes according to the defined rules. Several built-in notification services are provided for notification of conflicts due to concurrency control problems or missing tasks.

8 Conclusions

Information systems technology is expected—even stronger, is demanded—to play a key role in future software environments [5]. There is a strong request for *object management* technology going beyond file servers and that is the very nature of information systems. Yet, at least two requirements of engineering domains are not adequately covered by information systems technology: *group support* and *change management*. The approach presented in this paper is a step towards these objectives.

From the viewpoint of *group support*, the sharing process approach brings the consideration of the objects back into formalisms underlying conversation-oriented models. Sharing processes provide a schema to describe the consistent access of agents to tasks and objects. Suitable agents are human designers as well workspaces; participation covers contribution of designers to task executions in terms of semi-structured messages, as well as object transfers and change tracking in workspaces. *Task sharing* structures communication while *object sharing* defines the structure for collaboration. Both of them are brought together in a common framework by defining rules that associate dependent events in object sharing protocols with events in task sharing protocols, and vice versa.

From the viewpoint of *change management*, we have not only taken a different view of advanced transaction concepts (object sharing), but have also provided formal means to associate it with group-oriented extensions to software process modeling (expressed as task sharing protocols). Together with the use of a knowledge representation language, the sharing process model allows a more flexible definition and control of change types and their interrelationships than fixed "red-book" rule collections or pre-defined Petri nets. As an example, we mentioned the idea that the group could define different protocols for the change of central and of fringe components. This can be a starting point for improving process-oriented consistency and data quality in repositories.

On the other hand, experience in office automation has shown that formal consistency and enforcement of bureaucratic procedures is not everything. Conflicts are a valuable source of diversity and better long-term product quality—*inconsistency* must be managed with equal emphasis as *consistency*. The copy-merge paradigm adopted in ConceptTalk is intended to intensify collaboration in smaller subgroups. Instead of superficially avoiding conflicts by exclusive access rights, it emphasizes conflict recognition and explicit conflict resolution. To prevent lost work due to communication-less optimistic parallel work, very large groups should embed this protocol in a stricter global protocol.

The ConceptTalk prototype is only a first step in evaluating the potential of the sharing process approach for change management. As already shown in usage experiences with extensible tool kits such as NSE, the definition and enactment of specific change types, and their integration into coherent management policies for group-intensive work structures is a major challenge which will probably also require extensions of the underlying reasoning mechanisms and protocol compilation techniques to be computationally effective. Ongoing application experiments with the ConceptTalk prototype in software engineering, hypertext co-authoring, and industrial engineering contexts are further revealing a large number of practical requirements.

References

1. E.W. Adams, M. Honda, T.C. Miller (1989). Object Management in a CASE Environment. *Proc. 11th Intl. Conf. Software Engineering*, Pittsburgh, Pa, 154-163.
2. F. Bancilhon, W. Kim, H.F. Korth (1985). A Model for CAD Transactions. *Proc. 11th Intl. Conf. Very Large Data Bases*, Stockholm, Schweden, 25-33.
3. N.S. Barghouti, G.E. Kaiser (1991). Concurrency Control in Advanced Database Applications. *ACM Computing Surveys 23*, 3, 269-317.
4. S. Bendifallah, W. Scacchi (1989). Work Structures and Shifts: An Empirical Analysis of Software Specification Teamwork. *Proc. 11th Intl. Conf. Software Engineering*, Pittsburgh, Pa, 260-270.
5. P.A. Bernstein (1987). Database System Support for Software Engineering - An Extended Abstract. *Proc. 9th Intl. Conf. Software Engineering*, Monterey,Ca., 166-178.
6. J. Conklin, M.L. Begeman (1988). A Hypertext Tool for Exploratory Policy Discussions. *ACM Transactions Office Information Systems 6*, 4, 303-331.
7. M.E. Conway (1968). How do Committees Invent. *Datamation 14*, 4, 28-31.
8. C.A. Ellis, S.J. Gibbs, G.L. Rein (1991). Groupware—Some Issues and Experience. *Communications of the ACM 34*, 1, 39-58.

9. U. Hahn, M. Jarke, T. Rose (1990). Group Work in Software Projects - Integrated Conceptual Models and Collaboration Tools. *Proc. IFIP WG 8.4 Conf. Multi-User Interfaces and Applications*, Iraklion, Greece, 83-101.
10. B. Hartfield, M. Graves (1991). Issue-Centered Design for Collaborative Work. *Proc. IFIP TC8 Working Conference on Collaborative Work, Social Communications, and Information Systems*, Helsinki, Finland, 295-310.
11. M. Jarke, ed. (1991). *ConceptBase V3.0 User Manual*. Report MIP-9106, Universität Passau, Germany.
12. M. Jarke, J. Mylopoulos, J.W. Schmidt, Y. Vassiliou (1990). Information Systems Development as Knowledge Engineering: A Review of the DAIDA Project. *Programirovanie 17*, 1, Report MIP-9011, Universität Passau, Germany.
13. G.E. Kaiser, S.M. Kaplan, J. Micallef (1987). Multiuser, Distributed Language-based Environments. *IEEE Software 4*, 11, 58-67.
14. G.E. Kaiser, D.E. Perry (1987). Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution. *Proc. of the 1987 Conf. on Software Maintenance*, 108-114.
15. C. Maltzahn (1990). An Environment for Cooperative Development. Diploma thesis, Universität Passau, Germany (in German).
16. H. Müller, J.S. Uhl (1990). Composing Subsystem Structures Using (k,2)-Partite Graphs. *Proc. Conf. Software Maintenance*, San Diego, Ca, 12-19.
17. J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis (1990). Telos: A Language for Representing Knowledge about Information Systems. *ACM Transaction on Information Systems 8*, 4, 325-362.
18. D. Perry, G. Kaiser (1991). Models of Software Development Environments. *IEEE Transactions Software Engineering 17*, 3, 283-295.
19. C. Potts, G. Bruns (1988). Recording the Reasons for Design Decisions. *Proc. 10th Intl. Conf. Software Engineering*, Singapore, 418-427.
20. T. Rose, M. Jarke, M. Gocek, C. Maltzahn, H. Nissen (1991). A Decision-Based Configuration Process Environment. *Software Engineering Journal 6*, 3 (Special Issue on Software Process and its Support), 332-346.
21. W. Sasso (1991). Motivating Adoption of KBSA: Issues, Arguments, and Strategies. *Proc. 6th Knowledge-Based Software Engineering Conf.*, Syracuse, N.Y., 143-154.
22. I. Shy, R. Taylor, L. Osterweil (1990). A Metaphor and a Conceptual Architecture for Software Development Environments. *Proc. Intl. Workshop on Software Environments*, Chinon, France, LNCS 467, Springer-Verlag, 77-97.
23. R.G. Smith, R. Davis (1981). Frameworks for Cooperation in Distributed Problem Solving. *IEEE Transactions Systems, Man, and Cybernetics 11*, 1, 61-70.
24. R. Unland (1991). TOPAZ: A Toolkit for the Construction of Application-Specific Transaction Managers. Report MIP-9113, Universität Passau, Germany.
25. T. Winograd, F. Flores (1986). *Understanding Computers and Cognition*. Norwood, NJ: Ablex.