On Sharing of Objects in Concurrent Design

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

Bernhard Mitschang
Universität Stuttgart, Breitwiesenstr. 20-22
D-70565 Stuttgart, Germany
mitsch@informatik.uni-stuttgart.de

Abstract

Sharing data is an important aspect in distributed design environments and should be supported by an underlying system. Any synchronous access to data is conflict prone. Applying concurrency control and two phase commit are an option to be considered. But design processes also demand cooperation between the designers. Negotiation about actions on the product under design and the early exchange of preliminary results are crucial issues. Controlled data access by itself doesn't fullfil all the needs for cooperation. We will present a new approach that relies on a concept and system model which integrates concurrent activities by a common information space offering flexible protocols for cooperation on the shared objects. We will describe the customizability of the protocols to allow the approach to be adapted to different cooperative scenarios.

1. Introduction

Processes in concurrent engineering and design rely on cooperation between the different participants. This cooperation is necessary to establish and coordinate the tasks needed to fullfil the goal of the design process. Those tasks (also called design steps) - in which the complete process can be subdivided into - rely on the results, states and progress of concurrent, previous, and future tasks. Typically it can not be exactly planned beforehand which tasks are necessary, how the tasks depend on each other, and how and when those tasks have to interact. Thus it is necessary to offer mechanisms to support cooperation at runtime.

We already presented our activity model for coordinating design tasks in [3]. In this paper we will discuss data sharing as the second pillar of the ASCEND project (*Activity Support in Co-operative ENvironments for Design issues*). The situation we started from is given by non-integrated systems which should share data in order to coordinate tasks in a joint process (Figure 1). The difficulty is that the data to be exchanged is not under a common system control. That might cause inconsistent views, conflicts and errors. So our answer is to transfer the data to be shared under a common and adequately customized system control.

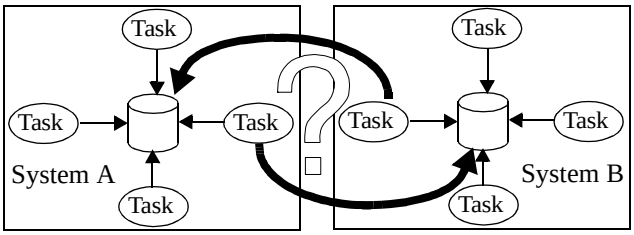

Figure 1. Data sharing for heterogenous systems

1.1. Process example

A new technical term for activities that integrate both, groupware as well as workflow is called *designflow*. To give an idea of our activity model, we describe the design of a new computer printer as a designflow (Figure 2).

First the general specifications for the printer have to be established: type (laser, ink), processor and chipsets, page format, OS platforms, pricing, and so on. This results into a specification document, which is stored in the shared information space (SIS). Also a preliminary list of hardware components and software modules (generic driver files, ...) to be used is collected and stored in the SIS.

We didn't detail the activity *Determine Specs* which might contain group meetings, and analysis of vendor documents. The first printer specification is used by the groupware activity *Design Coordination* to delegate the design of hard- and software. Those activities are further detailed, containing specific sub-activities. The hardware design activities have to work closely together, since they share e.g. the circuit layout, hardware specs and

components. Changes in the specs or layout can be propagated and when necessary negotiated. Similar, the software design activities have to have access to the current hardware design and specification in order to correctly write software for the printer. The software activities have also to interact on shared data (i.e. files, specs) as is typical for software processes. Tests are executed to validate the software and simulate the hardware (e.g. electro magnetic compatibility). The hardware simulation is described as a workflow since it is a automatable process. Finally the *Design Coordination* is responsible to decide if the design is finished.

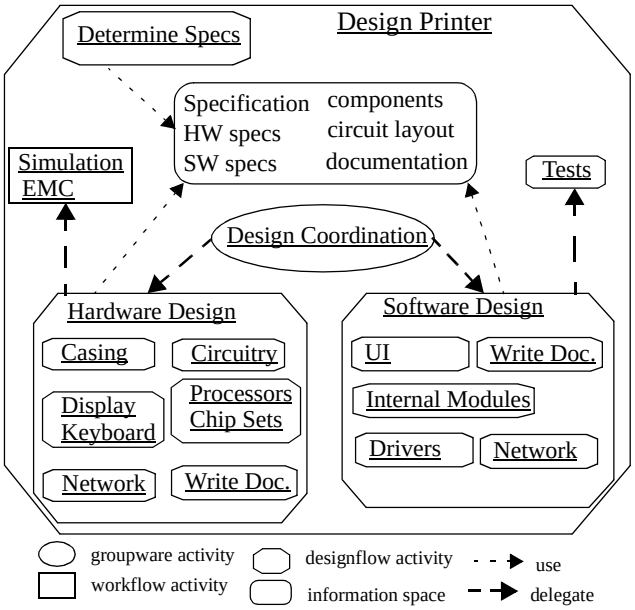

Figure 2. Designflow for printer design

The SIS in the example is important for the activities to exchange current design and preliminary results. Thus all activities can be aware of the work of others which they depend upon. For example, if the display is altered, the *UI* activity gets notified and can react to those changes. The notification is possible since the *UI* activity is registered on the (output) components.

1.2. ASCEND approach

A straightforward approach to support concurrent tasks is synchronizing them, like DB transactions or workflow systems do [5]. The processes we look at can be dynamic and require interaction and sharing of preliminary results. Thus they shouldn't be synchronized nor can be completely defined in advance. We actually don't want to serialize tasks anyway, since this prohibits any kind of parallelism. Instead we want to encourage cooperation, awareness and early data exchange which is typical for design and engineering processes. Additionally, we want to have system support for executing a process and individual actvities.

First we have a look at the activity model as part of the *ASCEND Designflow Model* (ADM). Our approach is combining groupware and workflow technology to comprehensively support structured and less structured parts of processes. The complete integration of existing groupware and workflow systems is not readily possible. We suggest an integration by encapsulating corresponding processes into ASCEND groupware and workflow activities, which can then be combined in ASCEND designflow activities (see [3] and Figure 3).

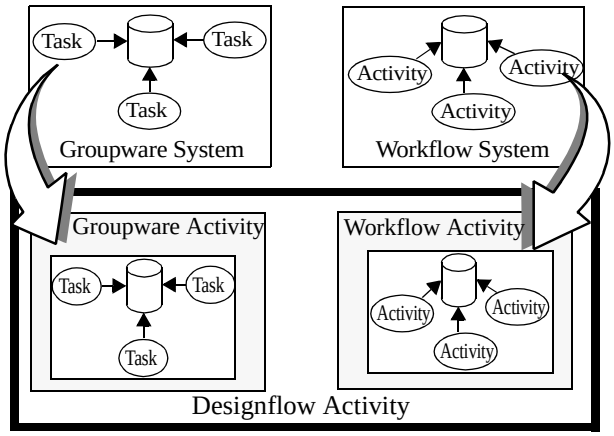

Figure 3. Combining different systems in a common activity model

The second part of the ADM regards data handling. The control of data access is a very well suited mechanism for coordination between concurrent tasks since data dependencies also translate into task interdependency. Those dependencies can be solved by concurrency control or additionally by cooperative interaction (e.g. negotiation) between tasks, respectively their actors (human users of activities).

The wrapping of the activities, as pictured in the above figure, realizes the integration into our designflow system. Our system also offers cooperation functionality like data sharing. For this the shared data has to be moved into the SIS (Figure 4). Logically the usage of those shared objects in the SIS is handled through the designflow system. I.e. the system guarantees that access to the objects is executed like the objects are natively stored through the designflow system. Resulting problems are mentioned in Section 3.

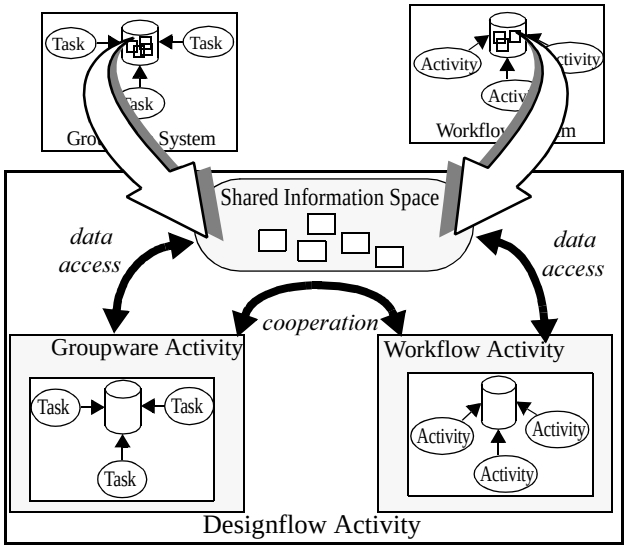

Figure 4. Transferring data into SIS

When examining objetcs to be used, one has to be aware that the objects often have dependencies between themselves, i.e. we have *complex object structures*. Sometimes complex objects are summarized as documents. Using this abstraction, one can think of a document being simultaneously edited by two users A and B. If the document is a single object, all actions on it are potentially in conflict. When the document can be "looked into"—i.e. it is separated into different objects like paragraphs or chapters—those sub-objects can be individually assigned and coordinated (e.g. [1]). Thus interference occurs less, when each user can only write to certain objects, which are distinct from the other user's ones. On the other hand if sub-objects are shared, awareness can be reached by making this fact known (e.g. displaying locked paragraphs). Another advantage is using object decomposition. Sub-objects can then be delegated to other tasks. This sub-object is exactly the part that the task has to work on, like delegating only the printer's case instead of all components. Of course it is possible to query any other sub-object of the complex object if it is necessary to access their information to complete a task, as would be necessary to know the dimension of keyboard and display to correctly define the printer's case. The sub-objects also have dependencies amongst them, like the printer's case has to be large enough to fit in the corresponding toner cartridge. We propose to use dependencies for objects in Section 2.2 for building complex objects to support such scenarios.

The last aspect discussed is the customization of objects. This is advantageous on two levels. First it is important to support requirement of different applications with different semantics of objects and even different object types. Secondly, as already discussed in [4], the granularity of objects is a crucial issue, which determines the degree of cooperation (and coupling). By using an object-oriented approach this should be fairly easy (Section 3). To finally use the objects with varying granularity and semantics, the access protocol also has to be flexible to adapt to different application semantics. Our answer to this is the use of conversation protocols in Section 2.3.

2. Design of the shared information space

In this chapter we discuss the concepts of the SIS. The information space comprises many aspects to support data exchange and cooperation. Since we use protocols to share objects, we can easily establish awareness and cooperation patterns. Also direct cooperation on the basis of interaction through protocols is possible.

2.1. Basic concept

In the ADM we call objects and activities entities, since they are somehow self-contained and can interact with each other. The basic aspects of the SIS comprise the information space itself containing objects and their dependencies, a communication layer for exchanging messages, and finally the protocols for communication between the entities (Figure 5).

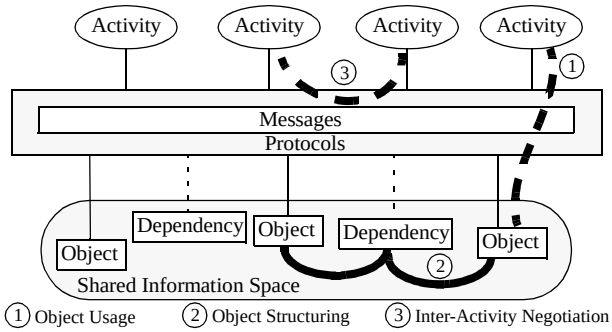

Figure 5. Communication and cooperation infrastructure for ASCEND

2.2. Objects and dependencies

The SIS is basically a simple name service, where all available objects and dependencies are being referenced. Thus objects to be shared have to be registered into this name service by some tool or application. Removal from the SIS is realized by unregistering the corresponding object. The object has to make sure that all connected activities and dependencies are then also disconnected.

Our approach to SIS can be characterized as being object-oriented. An object of the SIS implements at least the following structures and functions:

- user register
  Activities intending to use an object have to register on it. Thus depending on the access protocols implemented and applied to the object, users can be e.g. notified on object modifications caused by others.
- communication
  In order to communicate with an object, like modification, registration and so on, a receive function for messages has to be offered. Also the object itself has to have some kind of send function to communicate with other entities.
- privileges
  Typically the different users may have also different access rights to an object. Those have to be managed by the objects. Negotiation for privileges is executed via according protocols.
- dependencies
  Dependencies among objects are recognized.
- identifier
  An object has an unique identifier, and is stored by this in a name service.

All objects derive from the same interface, allowing dynamic handling of different objects types. Communication with an object relies on a known protocol (as opposed to fully dynamic conversations). This independency from the encapsulated resource eases the handling of different objects.

As explained before, dependencies are introduced for establishing complex object structures, e.g. to support task delegation scenarios with decomposed design items. The semantics of dependencies can be manifold. Currently we only look at “part of” and “is a” as dependencies, but whenever necessary, additional dependencies can be introduced. Of course the handling of dependencies can get in-efficient, when complex interactions between the objects have to be handled. It is up to specific implementations to find a compromise between functionality and efficiency. A dependency has to fullfil the following:

- object list
  Here all dependent objects are referenced. In case of e.g. a “part of” dependency, they can be distinguished regarding super-object(s) and sub-objects. By querying this list a sub-object may find its siblings and its super-object (vice versa for super-object).
- communication
  To realize interaction for dependencies and objects (e.g. notification) we also use the same communication interface like entities.
- identifier
  Like an object, a dependency is also stored in the name service by this identifier.

Typical scenarios involving dependencies include bill of materials, hierarchically structured objects, or objects with specific semantical relationships. Regarding our previous example, we may have a structure like the one shown in Figure 6. Further part-of relationships as for the chip set or the circuit board are existent, but have not been detailed. As one can see, the is-a relationship can be also very useful to express semantics for scenarios.

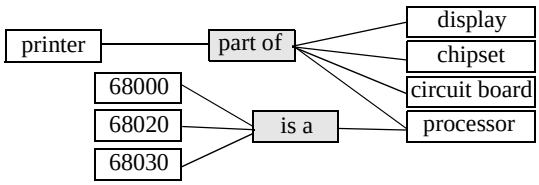

Figure 6. Object relationships for printer structure

2.3. Protocols

We picked aspects from conversation nets, speech acts [7] and agent technology to build powerful but simple protocols for interaction among the entities in our model. The advantage of combining those technolgies is the simple definition of protocols, comprehensibility and the possible deployment of a generic protocol engine, which can be customized to support different protocols. Our protocols realize the access, interaction and negotiation between the ADM entities. Thus cooperation patterns can be established and group awareness (e.g. querying all users of an object) achieved.

Figure 7 depicts a typical usage protocol, from the object’s view, as a simple example. First the object receives the registration request of an actvity (`usage`). Now the object has to decide, whether to accept the request or reject it (`reject/accept`). This decision can be delayed when the object decides to negotiate parameters of the request (`propose`), e.g. by proposing a read only privilege instead of an exclusive one. This negotiation can take some iterations until a final agreement or denial is reached.

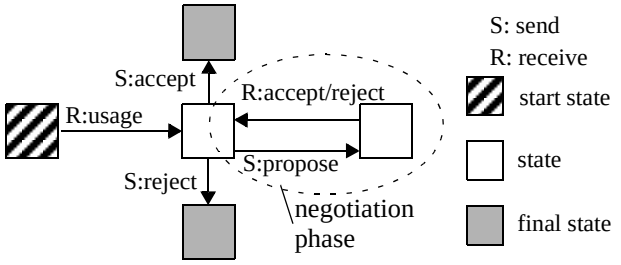

Figure 7. Usage request for SIS object

An object access protocol can look similar to the usage protocol. Instead of `usage`, the conversation topic would be for example `read`. But depending on the object type, an access protocol can become a lot more complex (Figure 8):

The activity E tries to modify object y (1). Since object y knows that it's in a tight coupled part-of relationship, it has to ask its super-object x for a permit to be modified (2). Object x's implementation requires registered activities to agree on changes, so it has to ask these activities for a permit itself (3). The result is transferred back to object y (4) and also made known to activity E (5). If the requested modification has been successful, the part-of dependency can then actively notify its other users object y about the modification (6) to ensure consistent views.

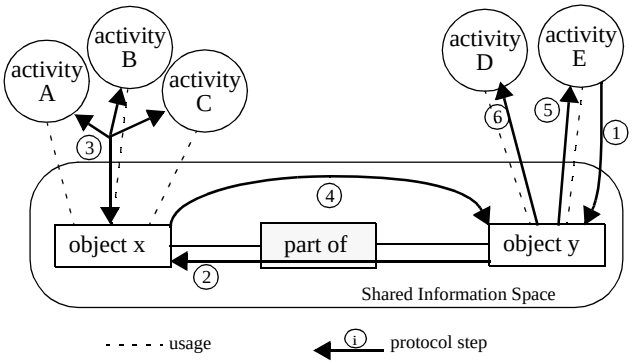

Figure 8. Complex access protocol

The above scenario is kind of a worst case of an access request (esp. if you take into account that step 3 could additionally require a further negotiation between activities A, B, and C to reach a common decision). But it illustrates what can be achieved through the combination of customizable protocols, dependencies, and objects. Generally one will not always implement dependencies and objects (and their corresponding protocols) as suggested by this scenario, but instead as is sensible for a specific application scenario.

3. Realization

We will now briefly describe how to realize the presented concepts and where customization is possible.

3.1. Object centered concept

The objects are a central concept to our approach since they are used to coordinate the work, raise group awareness, and support cooperation. Our implementation CASSY (*Cooperative Activity Support SYstem*) uses CORBA as middleware. This allows wrapping of proprietary systems and resources as well as the exploitation of existing services (events, workflow facility, ...). The groupware facility has been implemented by ourselves using CORBA. It offers typical groupware functionality like tasks, workspaces, groups, resources. Problems arise from the API supported by legacy systems, if they don't offer access to internal structures and functionality (e.g. IBM's FlowMark). In that case they can't be fully integrated by CASSY. E.g. if objects in the workflow system can not be locked over the API, CASSY has to make sure to lock the object for modification access outside of the workflow activity while the corresponding workflow is running.

ADM objects and activities are implemented as CORBA objects. Adaptibility of objects and dependencies is fairly easy by exploiting object-oriented concepts like inheritance. E.g. a simple part-of dependency implementation can just realize the storing of the referenced objects and querying them. When a tighter coupled approach is required, a dependency can be derived and additionally implement mechanisms like propagation of changes from one object to the other referenced objects.

3.2. Protocol realization

An adaptive object model alone is not sufficient for supporting manifold scenarios in design and concurrent engineering. Flexible protocols are as well needed to use the object model as suggested by the application scenario.

Being CORBA objects, the objects of the ADM have already some characteristics of active objects, which led us to look into agent technology. 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 JATLite package [6] as platform for agent communication. The message format has been defined in KQML [2]. KQML is very flexible and thus lets us define very sophisticated protocols. As can be seen in the screenshot (Figure 9) we implemented some

protocols and corresponding entities to test the definition of protocols and their usability. Currently we want to implement this approach into CASSY, because of the positive results from the prototype.

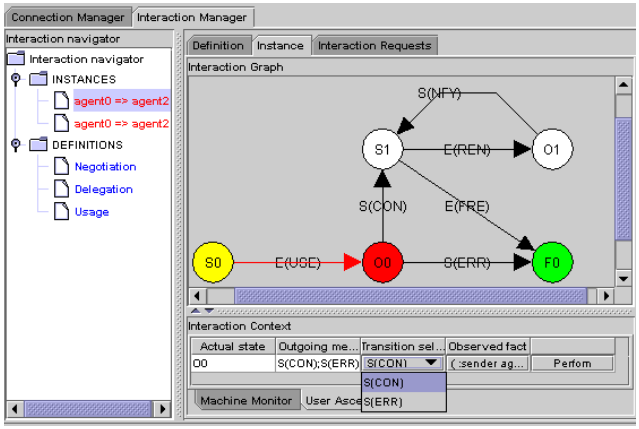

Figure 9. Screenshot of protocol agent

A further advantage of this implementation is that actors can also be integrated into protocols. The implementation can be used to offer an actor choices of answers (or requests) valid in a protocol (see list of transitions in Figure 9). Thus the proposed approach can allow direct interaction between actors and/or entities. If a protocol between entities reaches a point where human interaction is required a corresponding choice of messages can be presented to the actor. Through this we can integrate the actors more tight into the process.

4. Conclusion

Efficient design processes are characterized by the high degree of concurrency and interdependence of the single tasks or design steps. Those tasks are based on the structuring and execution of activities, and the usage of common data. We described a model to integrate shared objects in a common information space for concurrent activities. Our approach promotes cooperative techniques like awareness and negotiation through the actvity model and SIS combined with the protocols (e.g. delegation). This allows cooperation to take place between the actors and entities in our system model.

The adaptability enables one to customize the model to specific needs. The object protocols can be designed for many scenarios, even very sophisticated cooperation patterns. Object implementations can be suited to specific object types and exploit their full functionality through the generic message interface. Finally by the exploitation of dependencies, complex object structures can be established and functionally used. The degree of cooperation can be adapted to the scenario by accordingly adapting protocols and object granularity. Already the protocols we established realize a high degree of dynamic communication. We apply an implementation approach that is built on agent technology. In doing so the model can be even extended to dynamic cooperation interactions and ad hoc adaptability for newly arising situations.

Further work requires the implementation of exemplary protocols and object structures into CASSY to further validate the proposed concepts. Also it seems worthwhile to investigate whether those protocols could also be advantageous for communication between system services in CASSY, like the activity management or user client.

5. Acknowledgements

This work is partly sponsored by DFG grant MI 311/8. We also thank Erol Bozak for his work on the prototype implementation.

6. References

[1] Borghoff, U. M., Schlichter , J. H.: Computer-Supported Cooperative Work: Introduction to Distributed Applications. Springer, Berlin, 2000

[2] 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

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

[4] 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

[5] Gray, J., Reuter A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publ., San Mateo, CA, 1993

[6] Jeon, H., Petrie, C., Cutosky, M. R.: JATLite: A Java Agent Infrastructure with Message Routing. IEEE Internet Computing, IEEE, 200

[7] Winogard, T., Flores, F.: Understanding Computers and Cognition: A New Foundation for Design. Norwood, NJ: Ablex, 1986