Integration-Based Cooperation in Concurrent Engineering

Gerd Hillebrand Patricia Krakowski* Peter C. Lockemann Dietmar Posselt*
IPD, Universität Karlsruhe, Germany
{*ggh,krakowski,lockeman,posselt*}*@ira.uka.de*

Abstract

Concurrent engineering is a means to shorten product development times. Information systems supporting concurrent engineering must facilitate the exchange and unambiguous interpretation of product data from various design and manufacturing stages, the collaboration of geographically dispersed experts in a complex design process, and the use of diverse computing platforms and tools.

We describe our experiences in a joint project with the Department of Mechanical Engineering, building an integrated information system for concurrent engineering applications. The system is based on a single, integrated object-oriented data model and schema, the so-called product and production model, which covers all phases of the product life cycle. The model is implemented in a distributed and heterogeneous fashion using a CORBA-based object bus. On top of this bus, support for structured and ad-hoc collaboration is built around a uniform and flexible presentation layer. We motivate concurrent engineering and argue why we believe that it mandates a single, integrated schema. We then describe the overall architecture of the system and address various implementation issues, in particular object persistence, model evolution, presentation services and support for collaborative work.

1. Introduction

Today's development of new products takes place under immense time pressure: ever shorter technology time cycles lead to ever shorter product life cycles, global competition offers sufficient return on investment only to those who make it first to the market. "Time-to-market" has become the universal catchword to signify the pressure on engineers and sales forces.

Part of any time-to-market strategy is the shortening of development times. The traditional sequential process with its consecutive steps of submission of bids and tenders, product design, manufacturing planning, production scheduling, manufacturing organization and quality control must be replaced by a more concurrent one, where the steps are highly interconnected, where results are passed on in a yet incomplete state in order to initiate the first phases of the next step and where iteration takes place by providing corrective feedback to the earlier steps.

A process where many of the steps take place concurrently, albeit with relative delays, and where close interconnections and feedbacks must be observed, is commonly referred to as concurrent engineering. In concurrent engineering, many different experts from many different divisions of the same or different enterprises must collaborate and interact closely. Concurrent engineering can be successful only if the required collaboration is supported by information systems that reflect and enforce, among all persons involved, a common understanding of the objectives and the product of the engineering process, and that provide the participants at the right time with the right contents.

Since the experts involved, and hence the tools they employ, are spread across several divisions we have to expect that the common information system is distributed across a number of, in all likelihood, fairly autonomous component systems. Hence on a first glance, concurrent engineering seems to be a clear case for federated information systems, i.e., for information systems that tolerate a fair degree of semantic heterogeneity among the components, and facilitate interoperability between them by special means such as translation and mediation services. We claim, though, that concurrent engineering is under constraints so severe that it cannot afford to leave much room for terminological discrepancies, factual misunderstandings or ambiguities because these take too much time to resolve and leave too much of a risk if done by purely automatic means. Our thesis is that, instead, concurrent engineering necessitates agreement on a common, engineering-oriented product and production model that is shared and enforced by all component systems. Distributed information systems for concurrent engineering must be old-fashioned integrated systems with a common data model and database schema.

An integrated database with a shared database schema is a necessary, but by no means sufficient precondition for the

*Funded by the German Research Council (DFG) under project number SFB346.

successful integration of a collaborative engineering process. Engineering processes follow certain established patterns, hence one should be able to superimpose suitable process models on the database system or, more precisely, on its appearance through the product and production model. This would seem to bar a Computer Supported Cooperative Work (CSCW) approach because, although it relies on an integrated and shared document base, it is notoriously weak on imposing discipline on the cooperation, preferring instead to leave the evolution of the cooperation to the individual human players. At the other extreme, workflow management systems (WFMS) appear unsuitable as well because they impose tight discipline on the sequence of process steps and, consequently, require a fair amount of preplanning, which makes them less than ideal when it comes to dealing with unforeseen situations or with processes that are non-routine or ad-hoc by their nature, as are many engineering processes.

We conclude that in order to impose process models on a logically centralized and thus integrated information base, we have to find a middleground between CSCW and WFMS. As a second thesis, we claim that this middleground is provided by lifting the process model, along the lines of CSCW, onto the presentation layer of the common information system. Here the process becomes visible to the participating experts, and they may initiate, inspect and control its steps. We borrow from WFMS that each process step is associated with certain resources, such as computerized design, engineering, planning and analysis tools, and that these tools can pass information among each others via a common communication medium—in our case the integrated database.

Our third claim is that the integrated database must be supplemented by view mechanisms both at the schema and the presentation level. The product and production model provides a general and comprehensive core, while individual tools often require a highly specific view. Similarly, the presentation level provides a generic "desktop", which must be configurable according to the needs of each human user. By providing application-specific views (in the sense of traditional database views) at the schema level and user-specific views (in the sense of pluggable visualization components) at the presentation level, we allow engineers and tools to deal with locally flavored information. Efficient communication through the integrated database is still maintained, because both kinds of views are mapped to the core product and production model.

The system described in this paper is being designed and built in the context of SFB346, a joint research project between the Department of Informatics and the Department of Mechanical Engineering at the University of Karlsruhe. Funded by the German Research Council DFG, the goal of the project is to increase the efficiency of the product development and manufacturing process by suitable computer-integrated tools and infrastructure. After a strong initial focus on developing core object-oriented database technology, our current work emphasizes an open, distributed architecture. Hence the current, second generation of the system uses a middleware-based, peer-to-peer approach, whereas earlier prototypes were built using the client-server paradigm.

The paper is organized as follows. Section 2 surveys our product and production model and discusses how to organize a model of such size and complexity so that it can be mastered by the individual participants and tools. Section 3 gives an overall picture of the system architecture, which is based on an object bus providing information access and exchange in terms of the common model and a customizable, desktop-oriented user interface. Section 4 presents the user interface in some more detail and illustrates how views are used at the presentation layer to create individual perceptions of the PPM. Section 5 describes the cooperation support built into the system, which combines elements from CSCW and WFMS to support a process model along the lines of peer-to-peer communication among distributed participants. Section 6 discusses the interface between the object bus and persistent storage, and Section 7 deals with schema evolution and the issues resulting from supporting different versions of the schema simultaneously. These sections give a flavor of the technical challenges that have to be overcome in a flexible concurrent engineering environment. Section 8 reviews related work, and Section 9 concludes the paper.

2. The Product and Production Model

As pointed out before, in time-critical, distributed, cooperative product development it is imperative that the participants agree on the format and meaning of the data they exchange, and avoid the overhead that comes with interoperability options such as self-describing data [14, 19], Mediation [21], or agent technology. The recent trend in manufacturing industries towards standardized models such as STEP [5] demonstrates that a single, integrated data model and schema at the conceptual level seems to be the most economical solution.

For our research purposes STEP proves to be too unwieldy. Consequently, we designed the PPM, an object-oriented integrated product and production model suitable for describing the development and manufacturing process of mechanical components and assemblies. The model evolved in parallel with its larger cousin STEP—the Department of Mechanical Engineering also being an active contributor to the latter—and shares some of STEP's main features. However, because of its limited scope, it is significantly less complex and more manageable than STEP.

One of the major features of the PPM is that it encompasses the entire product life cycle. Given the many interactions between various phases of the product development and manufacturing process, it is essential that all data pertaining to a particular product, be it customer requirements, design specifications, or field maintenance records, are gathered in a—conceptually—single place. Despite its limited scope, the PPM is a huge model that must be intelligently structured in order to be amenable to collaboration and to facilitate its design and maintenance. The current version comprises some 300 object and relation types, grouped into 32 modules.

The model is divided into two layers. The lower layer provides a core model with concepts such as Product, Version, Property and so on. It is comparable to the Integrated Resources Layer of STEP. The core model is the one that establishes the common framework across all process steps. On top of it, an upper layer containing phase- or application-specific views is defined. It provides, for example, views tailored to requirements analysis, to principle, functional and shape design, to manufacturing and technology planning and others. The semantics of the upper layer is defined in terms of the core model. In the implementation, the upper layer is realized by a set of translation servers connected to the object bus, which convert requests issued against a particular view into requests against the core model. That way, the upper layer can easily be extended by defining a new view of the core model and writing a corresponding translation server.

Note that the schema-level views in the upper layer of the PPM are quite different from the views at the presentation level described in Section 4. The former define the format of data exchanged on the object bus and seen by applications, whereas the latter define the visualization of such data on a particular user's desktop.

3. System Architecture

The desired system architecture must be able to support all our three claims: the common information base modeled by the PPM, support for structured and ad-hoc collaborative processes, and a view concept both at the schema and the presentation level.

Our architecture has evolved over several project stages. Initially, the emphasis was on data integration: a common object-oriented database was shared by a number of complex engineering and planning tools. The engineering process under consideration at the time was strictly sequential, so there was no need for CSCW support or tight application integration. The PPM was used as a conceptual model, whereas the tools were implemented in terms of an underlying object-oriented model called GOM (Generic Object Model) [7], which defined the functionality of an ODBMS that guaranteed the persistence of PPM objects. GOM provided a persistent, object-oriented programming language (GOMpl) featuring both a descriptive data definition language (DDL) and an imperative data manipulation language (DML). A graphical schema editor allowed to enter GOM types in an OMT-like [15] notation and generate the corresponding DDL code. This code was the basis for implementing the behavior of objects as programs written in GOMpl.

The initial architecture afforded a smooth integration between the shared database and the programs accessing it, but tended to foster rather large, monolithic applications that did not lend themselves easily to the fine-grained, highly interconnected parallelism inherent to concurrent engineering. Moreover, because of its reliance on a single database programming language, it was difficult to integrate legacy code or off-the-shelf applications.

As the focus of the project shifted towards distributed and concurrent engineering processes, it became clear that a more open architecture based on fine-grained functional units was desirable. Moreover, an integrated, "desktop"-oriented user interface featuring view mechanisms and explicit modeling and monitoring support for cooperative processes became a necessity for smooth cooperation of workgroups.

These criteria are reflected in a new architecture, which consists of two major layers. At the bottom an open integration layer, called the object bus, connects distributed objects in a peer-to-peer fashion. Every object has a type specified in the PPM, which is now a runtime reality instead of the earlier merely conceptual tool. At the top an integrated presentation layer replaces the former collection of individual application GUIs. It employs a desktop metaphor in order to allow the more or less immediate interaction between collaborating engineers, and it offers views as individual perceptions of PPM objects to back the individual work.

Figure 1 shows the system architecture. It is strictly component-based: the presentation layer uses pluggable display components called views[1] to populate each individual desktop, while the object bus layer encapsulates primary data, metadata and application services as PPM objects.

The functional glue between the two layers and the distributional glue across each layer is realized by middleware. For our project, we have chosen *Orbix*, a commercially available implementation of the OMG's CORBA (Common Object Request Broker Architecture) interoperability standard [13]. CORBA combines the idea of trading

[1] These views at the presentation level are small programs (JavaBeans) that can display one or more PPM objects. As pointed out earlier, they are different from the application- or phase-specific views constituting the upper layer of the PPM. Unfortunately, the term "view" is deeply ingrained both in database and user interface terminology; with related, but not identical meanings.

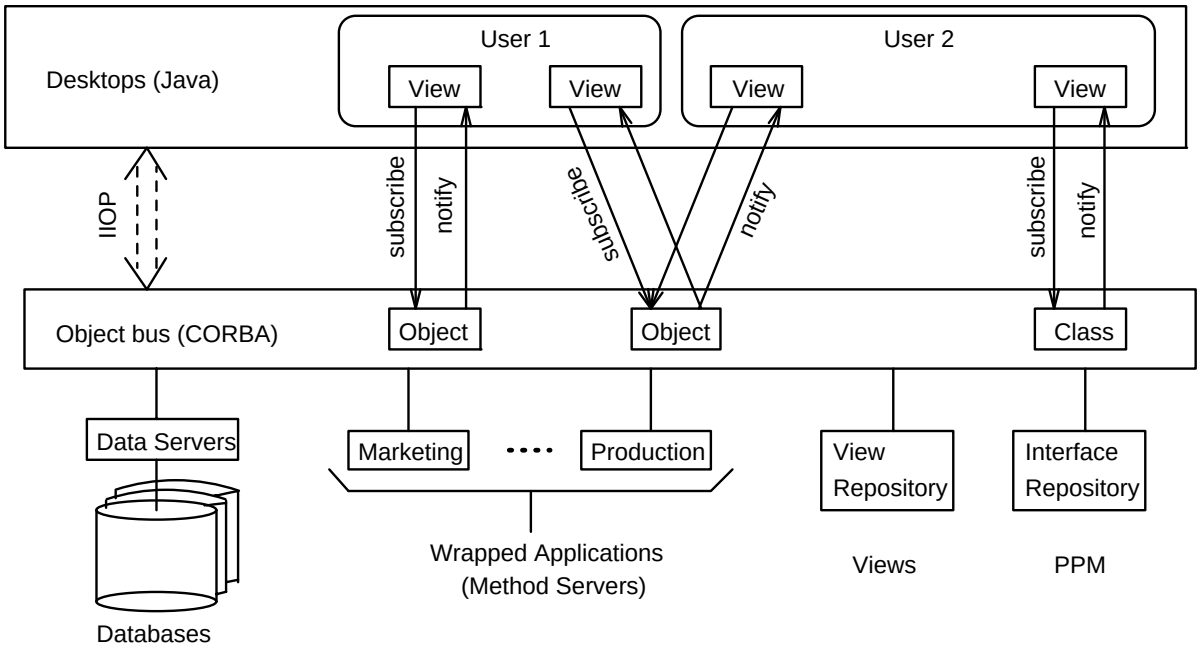

Figure 1. System Architecture

and brokering services to clients in a platform-independent way with the paradigm of object-orientation. Part of the CORBA standard is an interface definition language, IDL, for the specification of object interfaces. We use an extended version of IDL as the textual notation for the PPM. The Interface Repository and the Graphical Schema Editor (see below) translate back and forth between the textual and graphical presentation of the PPM. Using commercially available IDL compilers and object request brokers, the IDL specifications in the PPM can be compiled and implemented in any CORBA-supported language, e.g., Java, C++, Ada, Smalltalk, or even COBOL. Thus, even though the model is centralized and homogeneous, the actual implementation is distributed and quite heterogeneous.

The types defined in the PPM are managed by the Interface Repository, which ensures persistent storage of interface definitions. The Interface Repository is itself a collection of objects with IDL-defined interfaces, each object representing one PPM type. Thus, the metadata is held using the same data model and access mechanism as the actual data, which obviates the need for a separate schema browsing and manipulation infrastructure. Nevertheless, Interface Repository objects differ from ordinary objects in that modifications to the Interface Repository imply changes to the underlying database schema and perhaps object implementations. Therefore, a schema evolution mechanism is necessary in order to keep the running implementation consistent with the Interface Repository and to ensure that persistent data is smoothly carried over between model upgrades. This mechanism is discussed in Section 7.

Within the project, we use tools provided by third parties as well as tools that were developed during the early stages of the project. Consequently, we face the problem of reengineering legacy applications for use with the object bus. In the simplest case—when an application does not have a graphical user interface—only wrapping is necessary, which basically requires just the definition of an IDL interface. In general, though, engineering tools are heavily graphics-oriented and, even worse, large monolithic systems. Hence, the challenge is to break these systems down into fine-grained components, where ideally each component captures just one functional aspect.

The main responsibility of the presentation layer is to provide each user with a personalized desktop, which contains appropriate visual representations of those PPM objects the user needs for his tasks. Our approach is based on the Model-View-Controller [8] paradigm: The underlying PPM objects are regarded as *models*[2] with state and behavior, but no inherent visual representation, and separate *view* components are used to visualize models in different ways. A view has an associated *controller* component, which receives user input (from mouse or keyboard) and translates it into service requests to the model. Views and models are decoupled by establishing a subscribe/notify protocol between them—the view subscribes to the model, and when-

[2]There is again potential for confusion here: This usage of "model" originates from Smalltalk and is unrelated to database terminology, where a model typically denotes a schema.

ever the model's state changes, it notifies the view so that the view may update its appearance. Multiple views may be associated with a model to provide different presentations. For example, figures 2 and 3 show two different views of the same turning lathe: the first is geared towards a technology planner and presents various machine parameters in a textual fashion, whereas the second is geared towards the machine operator and uses a 3D image as a more intuitive representation. New views can be created for a model without changes to code associated with the model (the approach shows some similarity to the ANSI-SPARC three-schema approach of the Seventies [20]).

In our architecture a special component, the View Repository, maintains all defined views. Whenever the presentation layer is asked to display a specific PPM object, it interrogates the View Repository for an adequate view. The View Repository maintains, for each user and role, preference lists that specify how a given PPM object should best be presented. In accordance with these preferences, the View Repository selects a view and transmits it to the presentation layer, which then displays the view on the user's desktop. A generic view, which uses information from the Interface Repository to adapt itself to any kind of object, is available as a fallback in case no other view can be found.

To ensure platform independence and easy access via the World Wide Web, the presentation layer and the associated views are written entirely in Java. Communication between views and the underlying PPM objects utilizes the CORBA-standardized IIOP (Internet Inter-ORB Protocol), the same protocol that is used on the object bus itself for communication between PPM objects.

4. User Interface

The user accesses the system either locally or through a Web browser. In both cases, the Java classes comprising the presentation layer are loaded onto the user's workstation or PC and an initial login screen is displayed. The user identifies himself and the particular role in which he wishes to use the system—the notion of a role permits differently customized desktops for different functions—and this information is used to compile a list of PPM objects forming the initial contents of the user's desktop. Typically, this list contains the user's mailbox, current projects and tasks, favorite tools, and whatever other object the user wishes to have at his fingertips. The system does not attach any special semantics to the objects in the list; it merely asks the View Repository to identify a suitable view for each object, given the user's identity and role, and brings up these views on the user's desktop.

Once the login process completes and the initial views are active, the desktop presents the typical WIMP interface (windows, icons, menus, pointers) familiar from other desktops such as Microsoft Windows or Mac OS. Its look and feel, along with view properties such as colors or fonts, is set on an individual basis, directed by per-user preference lists held in the View Repository.

All interaction between the user and the system happens via views. Hence, an action typically has two phases: a navigational phase, in which the appropriate views are brought onto the user's desktop, and an operational phase, in which the user interacts with these views to display, modify, or control certain PPM objects.

Navigation is accomplished by following relationships between PPM objects, which map naturally into container/contained relationships between views. For example, the PPM description of a turning lathe consists of a root object containing references to various other objects such as parameter settings, cutting tools, maintenance events and so on. A view of the turning lathe typically displays some or all of these associated objects as well, using embedded subviews that are either just buttons or otherwise reduced in detail to keep the presentation manageable. Clicking on an embedded subview brings up a separate, full-fledged view of the corresponding PPM object. For example, the top four buttons of the turning lathe view in Figure 2 are actually embedded subviews of certain objects associated with the turning lathe; three of these subviews ("Leistungsdaten/Genauigkeit", "Werkzeuge/Spannmittel" and "Drehwerkzeug"—the system was designed in Germany) have been expanded and are visible as separate windows underneath. Moreover, the "Werkzeuge/Spannmittel" view again displays a list of subviews, one of which has been selected and expanded in the bottom-right window. In the 3D image in Figure 3, various subviews are incorporated as sensitive image regions: clicking on the die mount brought up the separate view at the bottom showing certain geometry parameters of the machine.

In addition to this kind of navigational access, two other access mechanisms are available: a customizable list of global access points available from a menu bar, and a search facility attached to every PPM type that can locate instances satisfying certain properties.

After the desired view has been opened, the interaction with the view depends very much on the facilities provided by the view. A simple view may provide editable text fields to change selected attributes of the underlying PPM object or buttons to activate selected methods. More sophisticated views may encapsulate entire applications such as a word processor or a CAD tool. In general, a view has exactly the same kind of access to the object bus as any other client, and it may use whatever navigation, attribute accesses and method invocations are necessary to accomplish its purpose.

The administration of the View Repository and the PPM is also done by means of suitable views. The View Repository, like the Interface Repository, is itself modeled in the

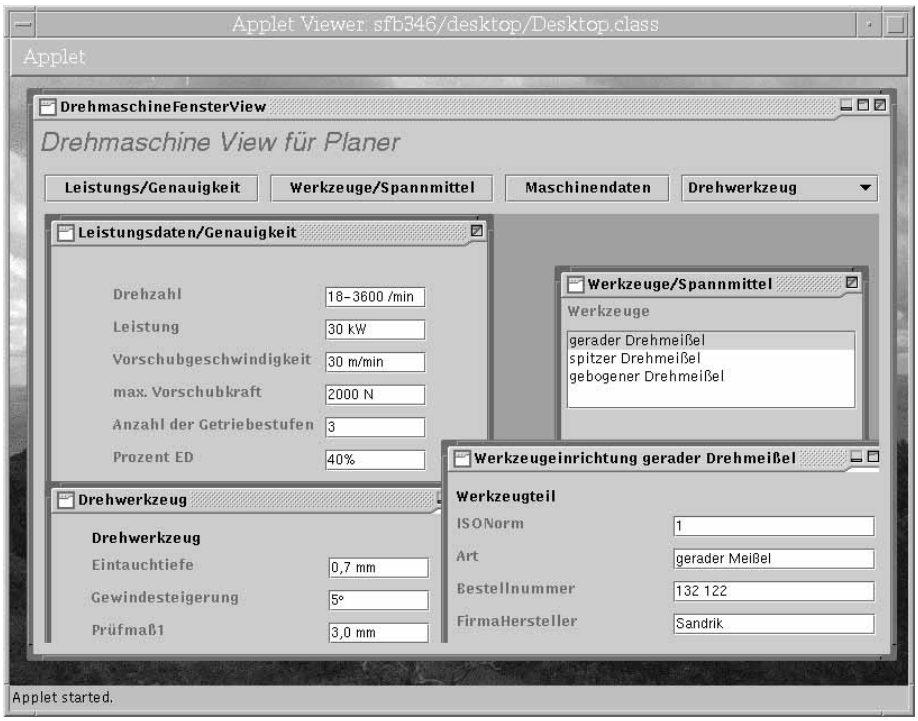

Figure 2. Planner's view of a turning lathe

Figure 3. Machine operator's view of a turning lathe

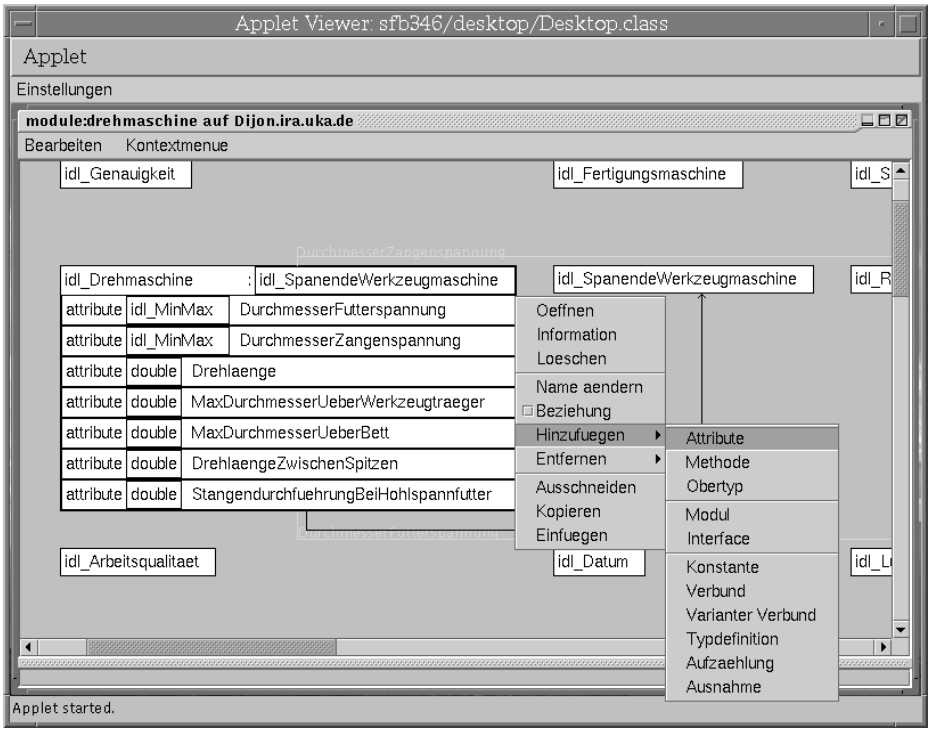

Figure 4. PPM maintainer's view of the type "turning lathe"

PPM and accessible through the object bus. Specialized views are available to access these administration objects. Each user can use so-called customizer views operating on the View Repository to adjust his personal desktop configuration. The administrator of the PPM has access to a collection of views operating on the metadata objects in the Interface Repository and can use these to modify the PPM and initiate code generation as described in Section 6. For example, a view of the PPM module describing, among other things, turning lathes is shown in Figure 4. This particular view contains a rather deep hierarchy of subviews (every white rectangle is a subview), corresponding to the nesting of metaobjects in the interface repository.

From the perspective of the presentation layer, these administrative views are by no means special—they are loaded from the View Repository and integrated into the desktop just like any other view. However, the View Repository and the Interface Repository also provide a bulk load facility so that the system may be bootstrapped from a set of configuration files.

5. Cooperation Support

The mechanisms provided by the system to support cooperative work fall into two categories: Support for more or less unstructured, ad-hoc collaborations and support for fairly well structured collaborations following a prespecified pattern. Mechanisms in the first category provide basic CSCW functionality and are mostly built into the system infrastructure, whereas mechanisms in the second category are workflow-oriented and rely on an explicit process model in the PPM with associated modeling, execution and monitoring components.

Many of the features often connected with "groupware" are already inherent in the system architecture. The PPM and the object bus sustain the fiction of a single, shared repository containing every piece of information relevant to the engineering process. Storage and exchange of documents, and more generally, any kind of PPM object, are handled transparently to the user—in fact, there is no real exchange, because only object references, not objects, are ever passed around. The views, through which human users perceive PPM objects, operate on the same underlying objects, and hence changes made by one user are automatically visible to all other users. For example, a shared group calendar or agenda is easily implemented by creating a persistent PPM object of the appropriate type and making it available to all members of the group. Every view displaying the object will be notified, through the subscribe/notify protocol, whenever the object changes state, hence group members will always see the current state of the agenda.

Various forms of communication between users can also

be conveniently realized using the desktop metaphor and the common object bus. For example, an arbitrary object may be placed into a user's input folder using drag-and-drop functionality supplied by the desktop. Combined with the notification mechanism, the recipient immediately detects the object in his input folder and may then click on it to start the appropriate view. Experiments are currently underway with videoconferencing: Double-clicking on a person's icon on a desktop initiates a video connection to that person's office. In addition, "virtual buildings" can be displayed by appropriate views, showing, e.g., the location of each office along with an indication of whether its occupant is currently willing to engage in any form of communication.

The support for structured, workflow-directed collaboration is based on a—currently fairly simple—process model in the PPM. Processes are partially ordered sets of activities, which may be either atomic steps or subprocesses. Activities have inputs and outputs, which may be connected by producer/consumer relationships to other activities. In addition, each activity has a pre- and postcondition, which are verified by the system before initiation and after termination of the activity, respectively. Every active process is associated with a container object, which collects the process-specific state of the objects participating in the process, in particular the inputs and outputs of each activity. When an activity is ready to be initiated, the system places the activity object together with its inputs into the input folder of the user responsible for it. The user may then display these objects on his desktop, perform whatever actions are necessary to complete the activity, and notify the system once the activity is finished, providing the values of the output parameters. Views are available to display and modify the activity structure of a process and to display the status of an active process.

The current, graph-based process model is best suited to well-structured processes such as traditional workflows. However, it is difficult to describe processes like collaborative product design, which have limited formal structure, yet follow certain patterns. We are investigating other means of description [10, 17], but the issue is still open.

6. Object Persistence

A PPM object, like any other object, contains state and implements behavior. The state must be kept in persistent storage, while the behavior is implemented by code that is shared across all instances of a specific type. Unfortunately, one cannot tell from the type of a PPM object what kind of state it contains. This may seem surprising, because in ordinary object-oriented models, the state of an object is modeled by its attributes. However, one must keep in mind that the PPM and the associated object bus are realized using CORBA, and that CORBA "objects" are really just handles to underlying implementation objects that provide certain operations specified in their IDL interface specification. In fact, an "attribute xxx" declaration in IDL is nothing but syntactic sugar for a pair of *get*xxx and *set*xxx functions, which may or may not be related to an attribute in the object's implementation. Thus, there is no a-priori reason why the attribute declarations in a PPM type should have anything to do with the state space of the underlying implementation.

However, some convention for declaring the actual state space of a PPM object must be chosen, if the system is to provide automatic persistence for PPM objects. The current prototype assumes, for simplicity, that the actual state space of the implementation of a PPM object is indeed given by the attributes in its PPM type. Put another way, this means that the result and side effects of any method call are completely determined by the parameters of the call and the values of the PPM attributes of the object the method is invoked upon. From a software engineering perspective, this may seem a poor choice, because it exposes the entire state of the object in its public interface. However, a separate access control mechanism at the instance level is needed anyway to implement per-user and per-object access rights, and the chosen approach does have the advantage of not requiring additional language constructs to describe the actual state space.

Given this assumption, persistent storage of PPM objects can be realized by providing *attribute servers* that implement, for each attribute of an object's PPM type, the get and set functions that the attribute declaration is converted into, using persistent storage. Because of the platform independence of CORBA, any implementation language and storage mechanism may be used, as long as the get function for a given attribute and object id reliably returns the last value set with the set function. In our system, attribute servers based on an object database system are automatically generated by a code generator, using information in the Interface Repository.

To facilitate the integration of legacy software and different computing platforms, but also for load balancing purposes, it is often desirable to split the implementation of a PPM object across several hosts. This means that clients hold a single object reference, but requests issued against that reference are directed to different hosts, depending on which method was invoked. For example, the attribute servers in our system are automatically generated and therefore implement only the get and set functions for object attributes, whose semantics are fixed. Other methods defined by a PPM type are implemented in separate servers, so-called *method servers*. For example, a method for computing the stress load of a mechanical part might be implemented by a method server that is actually a wrap-

per around a finite element analysis tool, whereas a method for estimating the manufacturing cost of the part might be implemented by a different server using a neural net. If both methods are associated with the same PPM type, clients holding a reference to the part object can invoke either method on the reference without being aware of the different implementations.

A single method server may implement any number of methods for any number of types. Since by assumption the entire state accessed by methods is captured in the attributes of the corresponding PPM type, method servers do not associate any state with object references themselves, but instead invoke attribute get and set functions on object references as necessary.

The distributed implementation of a single PPM type in several CORBA servers presents some technical challenges, since CORBA expects each type to be implemented in its entirety in a single server. The mechanisms used to overcome these difficulties will be discussed in a moment. However, for the client of a PPM object—and in fact the CORBA infrastructure itself—, the distributed implementation is completely transparent. Only the servers cooperating in implementing a PPM type know about each other and forward method calls and attribute accesses to each other behind the scenes.

To illustrate the distributed implementation of a PPM type, consider the following type definition (in IDL syntax, a type is called an interface).

```
interface i {
  attribute string a@s1;
  void m@s2();
};
```

This type definition declares an object type named i with two members: an attribute a of type string implemented in an attribute server named $s1$, and a parameter- and resultless method m implemented in a method server named $s2$. The @-syntax is not part of standard IDL; we added this construct to be able to name the servers participating in the implementation of a type. It is worth noting that these server names are logical names; they do not imply an affinity to a particular machine. The CORBA object request broker decides at runtime where to start servers based on its configuration tables.

From this type definition, two CORBA servers are automatically generated. From the perspective of the object request broker, both servers appear to implement IDL interface i, and the only observable difference is the logical server name. However, the two implementations are actually quite different. Server $s1$ is connected to an object-oriented database (ObjectStore in our case), whose schema, in this simple example, contains a single type i_db with a single, string-valued attribute a. For every PPM object of type i, $s1$ maintains a database object of type i_db, whose object identifier is embedded in the object identifier of the PPM object. Whenever an attribute access request arrives via the object request broker, $s1$ extracts the object identifier of the database object from the incoming PPM object identifier and accesses the database object to get or set its a attribute. However, when the object request broker asks $s1$ to invoke method m on a PPM object, $s1$ forwards the request to $s2$ by a judicious manipulation of the CORBA "envelope" of the request. The implementation of interface i in $s2$ is exactly the opposite: requests to read or write attribute a are forwarded to $s1$, whereas invocations of m are handled locally. Of course, the automatically generated code for server $s2$ cannot provide a real implementation of m since the code generator does not know what m is supposed to do. Instead a dummy implementation is generated, which must be manually replaced by code realizing the desired functionality. If this code needs to access attribute a, it uses the same forwarding functions that are invoked when an attribute access request comes in from the outside.

This multilateral forwarding technique can obviously be extended to any number of attribute and method servers. Except for the actual method implementations, all code, including the ObjectStore schema, is generated automatically from the online representation of the PPM in the Interface Repository. The attribute server also contains code providing a "factory object" for every PPM type, which allows for the creation, deletion and search of objects of that type.

To achieve robustness of actions one employs transaction models. Presently the only robustness we guarantee relates to individual requests: Every attribute request sent to the attribute server is executed as one short transaction. A chain of requests can be executed under ACID semantics if care is taken by the attribute server to execute all requests in the same thread as part of the same ObjectStore transaction. Using this technique, arbitrary ACID transactions can be realized on the object bus by assigning each object bus transaction a dedicated thread in the attribute server whose identifier is passed around as a transaction context. All requests arriving at the attribute server under the same transaction context must then be dispatched to the corresponding thread, which itself runs a single ObjectStore transaction. However, this mechanism has not been implemented yet.

7. Model Evolution

Business organizations and processes change often—more so than in the past. After all, concurrent engineering is a reflection of a much more dynamic production environment. Even though the PPM strives to be a general and comprehensive model of the product development and manufacturing process, it is to be expected that the model

will undergo frequent revisions and augmentations. It is therefore important to provide, as part of concurrent engineering support, schema evolution mechanisms that allow fairly general manipulation of the PPM while preserving the persistent data across schema changes.

There are essentially two ways of handling schema changes. One is the "big-bang" approach: The system is brought down, the old database is reorganized according to the new schema, and all CORBA clients and servers are modified and recompiled. Even in a research environment like ours, this approach is a major undertaking and only justified after extensive modifications to the PPM.

The other approach is incremental and does not disrupt the running system. It uses versioning to support multiple versions of a PPM type and the CORBA servers implementing the type. The persistent state belonging to a PPM object is shared across all versions of its type; however, the different versions may expose different parts of the shared persistent state.

The versioning approach works by making the clients of a PPM object responsible for selecting the correct object implementation. More precisely, if any program—view, CORBA server, or other—*uses* a particular PPM type in the sense of invoking operations of an instance of that type, then the client is responsible for directing these operations to a server implementing the same version of the type as used by the client. This is a major departure from the architecture as described until now, where clients of objects were always oblivious to the object implementations. However, application programmers can be shielded from this responsibility, because the required code can be generated automatically as illustrated below.

CORBA presents a remote object to a client by means of a *proxy object*, which is a client-local object of (essentially) the same type as the underlying remote object and which simply forwards all operations to the latter. The code for the proxy object is usually generated by an IDL compiler from the IDL specification of the remote object. When using versioned types and servers, however, the proxy code is modified to direct each operation to a particular server (identified by its logical name and version number), namely the server implementing the version of the type that was current at the time the proxy was generated. If different operations of the type are implemented in different servers, then the appropriate server is set in the proxy individually for each operation. The proxies are generated from information in the Interface Repository, which maintains the version numbers of types and servers and can therefore tell which server version implements a given type version.

By hardcoding the server name and version into the client proxies, it is guaranteed that when a client invokes an operation declared in some version of a type, the invocation is always handled by an implementation of the same version of the type. Thus no runtime type mismatch occurs. The load imposed on the system by keeping all versions of all servers accessible is not as bad as it may first seem, since servers are dynamically activated and not all servers need to be running all the time. Of course, the executables must be kept around.

Besides guaranteeing type safety, the system must also ensure that for a given object, the persistent state visible through different versions of the object's type is indeed shared. Of course, one first has to specify exactly which state is supposed to be shared. We adopt the following convention: Two attributes in different versions of the object's type share the same value iff they have the same name and the same type (in the type comparison, only type names are used and versions are not taken into account). In other words, if the PPM maintainer defines a new version of a type and adds to it a textual copy of an attribute declaration from an earlier version of the type, then he is telling the system that the attribute has the same semantics in both versions and that changes to its value made through one version of the type should be visible through the other version. In all other cases, attributes defined in different versions are independent. Thus, the persistent state of a PPM object is given by the collection of values for all attributes in versions of the object's type, subject to the sharing condition above. Hence the persistent state space of a PPM type never decreases as versions are added.

As new versions of a PPM type are created, the corresponding new versions of the attribute server face the task of mapping any newly introduced attributes onto the existing database schema. Since it is usually impossible to add new attributes to existing types in the database schema, the only solution is to define new *extension types* that provide space for the new attributes. From the very first version on, every database type must then provide for a *forwarding pointer*, which is used to chain extensions objects to existing database objects. Whenever an attempt is made to access an attribute of a PPM object whose associated database object does not yet contain an extension for the attribute, the necessary extension and any missing predecessors defined by earlier type versions are added to the chain. In the worst case, an attribute access requires the traversal of $n-1$ forwarding pointers in the database, where n is the number of the type version that first introduced the attribute. Therefore, a periodical reorganization of the database is still desirable, but this reorganization can be limited to the database and the attribute servers and does not affect the rest of the system.

8. Related Work

The integrated information system presented here falls under the broad label of "cooperative information sys-

tem" (cf. [2]), emphasizing group work and adaptability to change. However, it is somewhat unique in its strong reliance on the integrated model PPM, which sets it apart from heterogeneous federation architectures such as the ARPA I^3 Reference Architecture [4] and TSIMMIS [1] or database federation architectures such as [3, 9, 18].

The JTF (Joint Task Force) Reference Architecture [11] is conceptually related to ours, although a four-tier architecture was chosen. Based on a CORBA object bus, JTF layers generic services, applications and user environments. JTF integrates legacy data sources as coarse-grained objects, whereas fine-grained data objects are available in the form of *object webs*. Metadata is held in a schema server. The architecture does not specifically address issues of data and schema integration, although an integrated data layer seems to be implicit.

CoopWARE [12] is another integration architecture based on active database technology. A central information repository with an integrated schema provides data integration and persistence, while an ECA-based coordinator controls the interplay of a number of components acting on the repository. However, the system seems to be mainly targeted towards a single user interacting with a collection of tools.

The technical issue of interfacing a CORBA object bus to an OODBMS is also discussed in [6] and [16]. The former paper describes a framework developed by IONA Technologies to connect their Orbix product to the ObjectStore ODBMS. The latter paper (in German) is a tutorial providing background material on the various technical difficulties that must be overcome.

9. Conclusion

We set out with the hypothesis that smooth and efficient collaboration in concurrent engineering demands an integrated information system based on a single data model and schema. This assumption seems rather old-fashioned in a corporate IT world featuring large networks of heterogeneous and autonomous nodes, data being held decentralized in any number of formats, and change as a rule rather than exception. However, we have attempted to show that distributed object technology has come far enough to organize this apparent chaos into a highly flexible, orderly system of interoperating components. Starting from the integrated model, we introduced the object bus as a means of implementing the model in a distributed and heterogeneous fashion, and we discussed how to provide persistence, even when the model changes. We also showed how to support the collaboration of individuals with very different perceptions of the system, using a common desktop framework on which shared objects are presented in different views.

Beyond the basic functionality described in this paper, many open issues remain. Most notably are:

- Performance—the split into attribute and method servers described in Section 6 works fairly well for most engineering applications, because human speed is the limiting factor, but for compute-bound applications, a mechanism is needed to cache state in client proxy objects. Also, object replication and migration need to be addressed.

- Advanced transaction support—ACID transactions are not suitable for long-running, cooperative design processes. The system should support cooperative transactions and versioning of data.

- Process models—suitable description formalisms for ill-structured, cooperative design processes must be found, and made accessible to non-technical users. Moreover, an attempt should be made to explicitly model and record the goal of a process or collaboration, so that the system knows why certain things were done.

References

[1] S. Chawathe et al. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proceedings of the 100th Meeting of the Information Processing Society Japan (IPSJ)*, Tokyo 1994, pp. 7–18.

[2] G. De Michelis et al. Cooperative Information Systems: A Manifesto. In: M. Papazoglou and G. Schlageter (eds), *Cooperative Information System: Trends and Directions.* Academic Press, 1997.

[3] D. Heimbigner and D. McLeod. A Federated Architecture for Information Management. *ACM Transactions on Office Information Systems*, **3** (1985), pp. 253–278.

[4] R. Hull and R. King (eds). *Reference Architecture for the Intelligent Integration of Information.* Version 1.0.1, Program on Intelligent Integration of Information, ARPA, May 1995. Available via http://isse.gmu.edu/I3_Arch

[5] International Standardization Association. *Standard for the Exchange of Product Data.* ISO Standard 10303.

[6] IONA Technologies. *Orbix + ObjectStore Adapter.* White paper, 1993. Available online via http://www.iona.com /support/whitepapers/oosa

[7] A. Kemper et al. *Das GOM-Handbuch.* Technical Report 25/92, Universität Karlsruhe, 1992.

[8] G. Krasner and S. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, **1** (1988), pp. 26–49.

[9] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of Autonomous Databases. *ACM Computing Surveys*, **22** (1990), pp. 267–293.

[10] P. Lockemann and H.-D. Walter. Object-Oriented Protocol Hierarchies for Distributed Workflow Systems. *Theory and Practice of Object Systems*, **1** (1995), pp. 281–300.

[11] C. McKenna and R. Hayes-Roth. *Introduction to the JTF Reference Architecture.* Available online via http://www.teknowledge.com/JTF/jtf_arch_intro.doc

[12] J. Mylopoulos et al. A Generic Integration Architecture for Cooperative Information Systems. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems*, 1996, pp. 208–217.

[13] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* OMG 1995. Available online via http://www.omg.org/corba

[14] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the Eleventh IEEE International Conference on Data Engineering*, 1995, pp. 251–260.

[15] J. Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[16] C. Schmauch and B. Waibel. Warten auf den "Object Database Adapter". *OBJEKTspektrum*, January/February 1998, pp. 55ff.

[17] R. Schmidt. Component-Based Systems, Composite Applications and Workflow-Management. In: G. Leavens and M. Sitaraman (eds), *Foundations of Component-Based Systems Workshop*, Zurich, Switzerland, September 1997, pp. 206–214.

[18] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, **22** (1990), pp. 183–236.

[19] J. Thierry-Mieg and R. Durbin. *Syntactic Definitions for the ACEDB Data Base Manager.* Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge, CB2 2QH, UK, 1992.

[20] D. Tsichritzis and A. Klug. The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Management Systems. *Information Systems*, **3** (1978), pp. 173–191.

[21] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, March 1992, pp. 38–49.