Agent-Based Tool Integration for Distributed Information Systems [*]

M. C. Norrie and M. Wunderli
Institute for Information Systems,
ETH Zurich, CH-8092 Zurich, Switzerland.

Abstract

We present an approach for the integration of existing tools into higher-level application systems such as is required for workflow management, computer integrated manufacturing and computer-assisted software engineering. Tool integration is achieved through the introduction of agents which monitor and control the operation of the tool and provide an abstract view of the local data and operations. Then the management of data is distributed among the local tools and a central coordination system. Agent functionality is described in terms of both a general architecture and an examination of two specific agents — one for a revision control system and one for a CAD system. Following this, we present a general scheme for agent construction based on tool classification.

1 Introduction

Advanced application systems for business processing, computer integrated manufacturing (CIM) and computer-assisted software engineering (CASE) are all concerned with the integration of new and/or existing tools to achieve the coordination of high-level user and application tasks. A variety of technologies have been proposed to support such systems and these include workflow management systems [GHS95], repository systems [BD94] and product data management systems for CIM [CAD, She]. These technologies vary in terms of their intended functionality, application domain and approach to coordination. However, what they have in common is the necessity to interface to existing tools to gain the required level of control over their operation and data.

Many of these systems assume that the information flow that controls coordination is either based on email messages or a shared database. In the former case, it is assumed that users notify some form of central coordination system when certain actions have been completed and, as a result, other users will be requested to perform certain other actions. In this way, the tools are actually integrated into the system via the users. This approach has been adopted in some workflow management systems (see [GHS95] for an overview of existing products).

Alternatively, a central integrated database may be used to store all data objects and also rules which can trigger the system operations necessary for coordination. This approach is used in many repository-based systems for CASE [BD94]. In such cases, it is usually assumed that the tools are open systems to which external control and data access are provided. This may be the case with tools developed as part of such advanced application systems or according to new interoperability standards such as CORBA [OMG91] or STEP [STE92]. However, many existing tools, such as CAD systems, are large, complex systems which support limited forms of external access and cannot be

[*]Proc. 8th Int. Conf. on Advanced Information Systems Engineering (CAiSE'96), Crete, Greece, May 1996

easily replaced. Data is stored locally, usually not under the control of a DBMS, and using internal formats that frequently vary between system versions.

Current product data management systems [CAD, She] can be considered as combining these two approaches in the sense that data is stored in a central integration database, but the information and control flow between this database and the tools is performed through the users by means of explicit check-in/check-out mechanisms.

In this paper, we address the specific issue of how to integrate existing tools into distributed information systems in such a way that the information and control flow is automated as far as possible. We achieve such integration by means of tool-specific agents which monitor their operation and can invoke local operations.

We are not concerned here with a particular architecture or application domain. Most of the discussion is therefore independent of whether data resides only locally or is stored in a central integration database. However, we note that the approach described here has been adopted in a system for CIM tool integration which stores only coordination data in a central repository. Details of this coordination approach are given in [NW95, NSSW94, NWM+95].

In section 2, we discuss the need for such agent-based tool integration and the general requirements for such agents in terms of the functionality they must support. A general agent architecture is presented in section 3. In sections 4 and 5, we give overviews of two specific agents that we have developed. The first is a relatively simple agent which was developed for the revision control system, RCS. The second agent is for the CAD system, Pro/ENGINEER [Par93], and it exhibits many of the features typical of agents required for the integration of existing, complex design tools. Section 6 examines the various categories of tool architectures with respect to the required integration approach and effort and presents a general tool classification scheme. The resulting classification scheme is useful, not only in understanding the various issues and approaches, but also in enabling someone to construct an agent for a particular tool by identifying similar tool architectures and reusing implementation technique and code.

2 Agent Requirements

We assume an advanced application system built on top of existing tools which store and manipulate local data. Coordination of activities is achieved through some form of central coordination system which manages certain metadata and data. The extent to which data is managed, or replicated, centrally is not of importance to this discussion. Likewise, the precise nature of the coordination system and the forms of coordination supported is not significant here. Rather, we focus on the issue of how the coordination system interfaces to the tools in order that it knows that certain tool activities have been performed and, further, can initiate local actions.

The interface between a tool and the coordination system is realised by an agent as shown in figure 1. The two main tasks of an agent are to monitor the tool and keep the coordination system informed of any operations of global interest, and, to ensure that the tool performs operations necessary to the coordination process. It is not always the case that the tool operations can be invoked automatically as it may be necessary for a user to be involved in taking decisions or supplying information. In such cases, a message may be sent to the relevant users requesting that they take some action. However, it should not be the case that all tool operations have to be initiated by a user as is the case in some workflow management systems where all information and control flow is by means of email to users.

The agent must provide an abstract view of the tool's data and operation that corresponds to that of the coordination system. For example, for a CAD system, the abstract view of data and operations would be at the level of design objects and save operations on these objects rather than at the lower

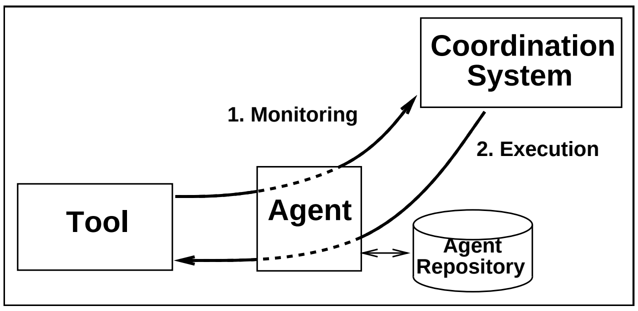

Figure 1: Agent main tasks

levels such as individual edit operations on a drawing. Likewise, the coordination system may not require knowledge of the object structure and value but rather only of its existence and relationship to objects operated on by other tools. Thus, for coordination in a CIM system, it may be required to know that a particular design object is related to a particular document object, but it is not necessary to know about the geometry of a drawing.

The abstract view of a tool's data and operation is provided by a mapping between the model of the coordination system and that of the tool. This mapping information is stored in the agent's repository. The coordination process must be subject to transaction management to ensure global consistency. The agent's repository therefore also stores log information about the coordination process and possibly even about the local tool data and operations in order that certain recovery actions may be supported.

The abstract view, as well as the model of the coordination system, are expressed in terms of a global data model. In the examples of this paper, we assume the semantic model NIAM [VB82] as the global data model. However, the general approach is independent of a specific data model.

3 Agent Architecture

In order to encapsulate tool-specific aspects into distinct components of an agent, and for the sake of reusability, we have split the agent architecture into modules. A schematic overview is shown in figure 2. In this section, we describe each of the four modules.

3.1 Communication module

The communication module deals with the input and output of messages between the agent and the central coordination system which would have a similar communication module. In the systems we have built, the communication module is based on a socket library as is available on major operating systems such as Unix, MVS, MacOS and Windows. The advantage of choosing such a low-level of communication is that it ensures independence of both the nature of the tool and the platform on which it runs.

The communication modules exchange information by means of a network transfer language that specifies the format of messages. A message comprises a header part and a data part. The header gives information such as the message source, the associated global and local transaction identifiers and the requested operation. The data part consists of any data objects to be transferred. Messages may be either control messages where the requested operations are related to transaction

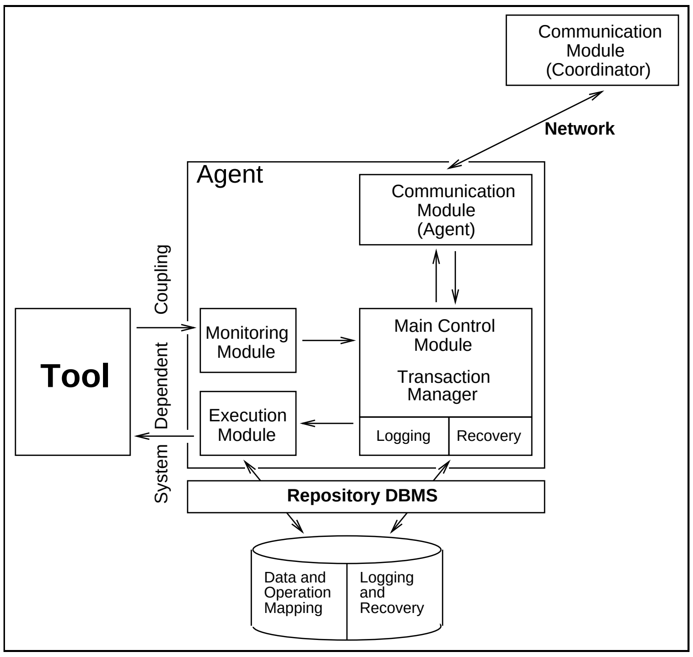

Figure 2: General agent architecture

management or data messages where the requested operations are related to the creation, deletion and update of data objects.

3.2 Main control module

The main control module of an agent schedules local operations initiated by a tool and requests from the central coordination system. The operation and complexity of this module varies greatly according to the nature of the tool, especially in terms of the notions of atomicity and concurrency that it supports. Generally, there are two fundamental categories of tool systems. One category consists of those such as CAD systems which have low concurrency, long-running transactions and frequently a check-in/check-out mechanism for control of shared access to data. The other category consists of those with high-concurrency and short transactions such as might be found in a resource management system; these systems are often implemented on a DBMS.

In the former category, a simple scheduler will suffice whereas, in the latter case, the scheduler must be more complex to ensure efficiency of operation. For example, for a CAD system, a scheduler may simply serialise incoming requests and interact with the user to ask for the release of locks on objects in order that the processing of these requests may proceed.

3.3 Execution module and monitoring module

The remaining two modules are the monitoring module and the execution module which are responsible for the monitoring of tool operations and initiation of externally requested actions, respectively. It is these modules that map between tool data and operations and those of the centralised coordination system. Their operation is very specific to the tool architecture. Thus, while the communication and control modules can be generalised, the execution and monitoring modules are tool specific. However, by recognising similar classes of tools, it is possible to reuse design, and even code, in the development of particular agents.

As stated, the monitoring and execution modules require a mapping between tool data and operations and those of the central coordination system. The overall coordination of the distributed information system is expressed in terms of a basic set of operations on objects such as *insert*, *delete* and *update*. These operations are interpreted locally by mapping them into a number of tool specific operations. Note that these operations may include requests and notifications to users as well as the direct invocation of tool operations. For example, the update of a new design document by a CAD tool may ultimately result in an email message to another user requesting that they create a corresponding update in the design documentation using a text editing tool. Such a user notification, however, should always be coupled with a monitoring component which detects whether a user has at least done *something*. Whether he did the right thing, of course, cannot be checked by the tool's agent.

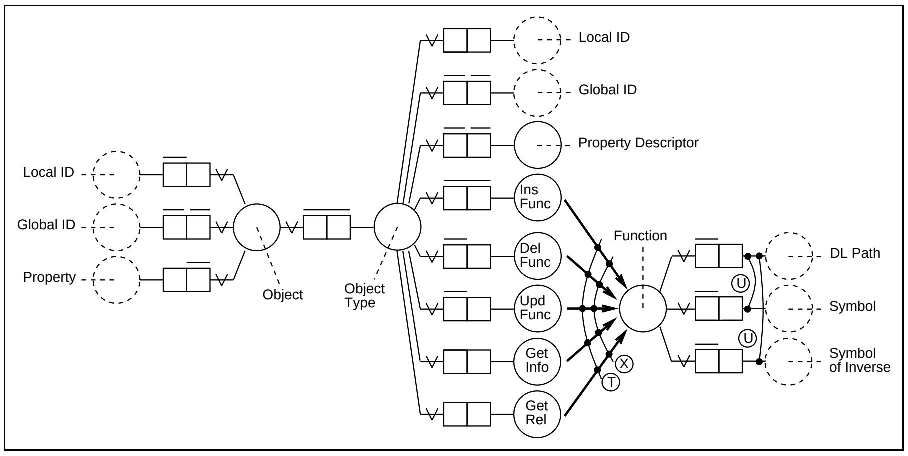

Figure 3: Metamodel for data model mapping

Figure 3 is a NIAM representation [VB82] of the agent's metamodel for this mapping. The metamodel shows the mappings between the global and local objects and object types. Solid circles represent NIAM object types, dashed circles stand for attributes. Attributes and object types are connected by the same construct, the NIAM fact type, which is graphically represented as a box. Cardinality constraints are imposed on fact types by an existing or non-existing $\vee$ (lower boundary 1 resp. 0) and an existing or non-existing bar (upper boundary 1 or *). For the sake of simplicity, we have omitted here the attributes associated with certain object types, e.g. `Property Descriptor`.

Global operations, such as *update* on object types are represented in the metamodel as object types describing the location of their implementation. We use the concept of dynamic linking and loading (DLL) to avoid the hard-coding of implementation of the global functions; the functions are dynamically loaded at the start-up time of an agent, thereby permitting flexible customisation. The function DDL paths and symbols are stored in the agent's repository.

Additionally, we have inverse functions corresponding to inverse global operations on the object types. These inverse functions implement compensating actions in the event that operations stemming from requests from the central coordination system have to be undone because a global coordination operation failed.

The mapping between global operations and local functions also provides operations to support the agent's task. The *GetInfo* operation allows it to retrieve information about objects from the local tool dataset in order that it may, for example, be included in the data part of messages and possibly used in other tools. The *GetRel* operation is used to get information about the relationships in which an object participates in order that insert or delete operations on objects can cater for any such dependencies and ensure that local consistency is not violated.

The mapping between local and global operations is directly implemented in functions of the monitoring module, and therefore is not visible in the meta schema. However, the concept of dynamic loading and linking (DLL) is applied too to avoid hardcoding of monitoring functions.

Since the monitoring and execution modules are tool specific, we discuss two particular instances in the following sections. The first is for the Revision Control System, RCS and the second is for the CAD system Pro/ENGINEER [Par93]. These two agents illustrate the differences that can arise in both tool characteristics and, correspondingly, the complexity and detailed architecture of their agents. Following on from the description of these two specific agents, we generalise to consider the various forms of agents corresponding to the various classes of tools.

4 RCS Agent

The revision control system, RCS (produced by the Free Software Foundation), supports shared access to documents by means of version management together with check-in/check-out mechanisms.

RCS is in fact a collection of small application programs which together form the complete system. Each application program performs a specific task such as checking-in a version or displaying log tables. These application programs can be used either directly by a user in interactive text mode, or, by other higher-level application systems, such as a document management system (e.g. Framemaker) in batch mode.

A tool, such as RCS, implemented as a number of user-task-specific programs and with a batch interface is amenable to the use of wrapper programs to realise the monitoring and execution modules of an integration agent. This is illustrated in figure 4. Each wrapper program has the same interface as the original application program, but, in addition to performing the user-task by means of a call to the original program, they perform the monitoring task of the agent. In this way, information flow from the tool to the agent is guaranteed and is at the logical level of user operations. The execution module of the agent is easily realised by directly calling the programs.

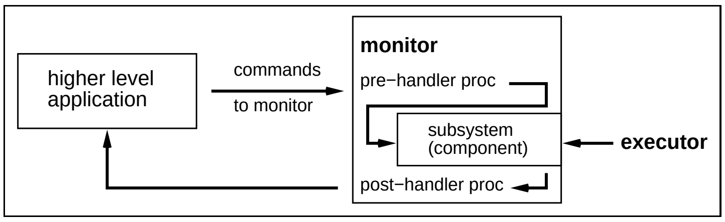

Figure 4: Monitoring and execution by wrappers

Concurrency of operations on files controlled by RCS is managed by the RCS system itself by means of lock files. However, atomicity is more of a problem as it is possible to lose the whole version tree of a document due to a system failure. We therefore regard RCS as not providing atomicity of operations.

We now describe in detail how the RCS agent realises the abstract interface and the monitoring and execution modules.

4.1 RCS abstract interface

As RCS files are plain UNIX files, they reside in the UNIX filesystem. The RCS files of our example scenario are located on the account `cimdoc` in folders grouping them according to their purpose as illustrated in figure 5. Multiple occurrences of a file in more than one folder is implemented using links, however this is not supported by the RCS system itself and has to be done by an account administrator.

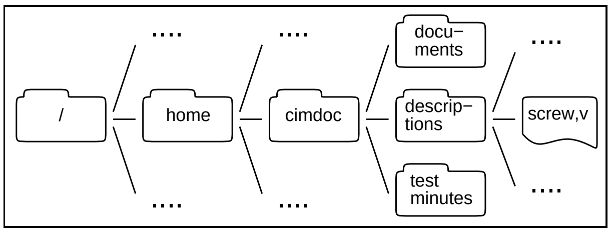

Figure 5: The RCS document management system's schema

The description of the local data in terms of the global data model is quite straightforward as shown in figure 6: The mapping between the local and the global description of object types is a 1:1 mapping. The abstract interface models a subset of the RCS system's data on each file representing attributes considered to be of global importance. These are the attributes which will be sent in the object information part of a request to the coordination system and which are required by other tools.

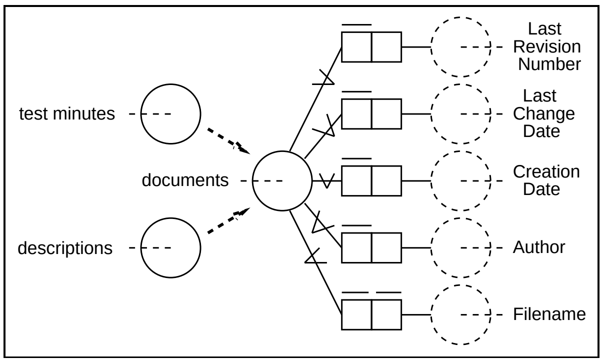

Figure 6: RCS abstract interface

Although the notion of a transaction is not present in the RCS system, a call to an RCS component can contain more than one RCS file as its parameters. In order to maintain this grouping in the coordination system, we group these multiple operations into a single (sub-)transaction. The mapping of the most important RCS operations to transactions is shown in table 1.

It is important to note that most tools have no transaction support. Such a feature normally has to be provided by the agent of a tool, a fact which is reflected in the state transitions of a user-involved and a coordination-system-involved (local) subtransaction. We will show this in the next subsection.

tool operation	induced transaction
co (check-out read)	snapshot operation, not relevant to coordination
co -l (check-out write)	start of (update) transaction
ci (check-in)	end of transaction, send update request to coordination system
rcsdel	transaction requesting the deletion of one or more RCS files
rcsinit	transaction requesting the creation of one or more RCS files

Table 1: RCS operations and their impact on the coordination system

4.2 Monitoring module

Since the monitoring is based on wrapper programs around the tool operations `ci`, `co`, `rcsdel` and `rcsinit`, each tool operation (program call) can be mapped to a single transaction for which confirmation has to be requested from the coordination system.

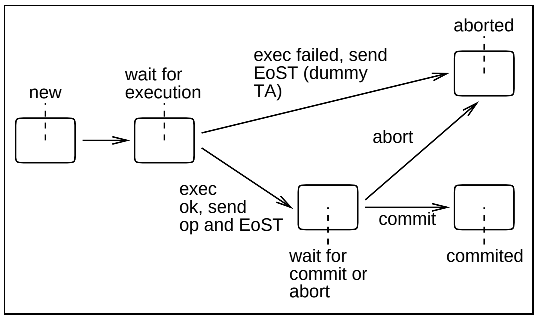

Figure 7: Performing a user operation

Figure 7 shows the possible state transitions of a transaction corresponding to a user operation. A subtransaction is started before the tool operation takes place in order to prepare the global transaction and the transaction associated logging. We then wait until the tool operation has completed successfully before sending the complete request to the coordination system (EoST: End of SubTransaction). If the tool operation fails, an empty transaction is sent which does no harm. If it succeeds, we are sure that we are really in a ready-to-commit state and therefore are really authorised to send such a request.

4.3 Execution module

The implementation of the execution module of the RCS agent is straightforward. In the case of delete and create requests, the corresponding operations `rcsdel` and `rcsinit` can be called by the execution module in the same way as they would be called by a user and therefore can be handled automatically. However, in the case of update requests, we have chosen an interactive approach. An email message is sent to the last author of a document requesting the update. If this user then performs a check-in operation (`ci`) of the file for which the update was requested, the requested operation is regarded as being fulfilled.

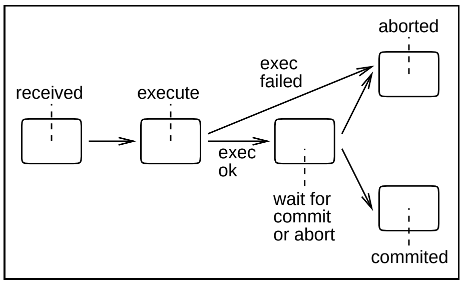

Figure 8: A request received from the coordination system

Figure 8 shows the state transitions of a request received from the coordination system. The agent starts a new transaction if it receives such a request. It then waits until the execution module reports success or failure of the requested operation and afterwards goes into the state 'aborted' or 'wait for commit/abort'. The latter wait state is left depending on the answer of the coordination system as to the outcome of the associated global transaction.

5 Pro/ENGINEER Agent

Pro/ENGINEER [Par93] is a modern CAD system both in terms of its design paradigm based on feature modelling and its extensibility. The system stores its data in files with each design version stored in a separate file. Therefore, while it does not support atomicity directly, its storage of version data does facilitate support for atomicity as provided by the agent. The system does not control concurrent access to data, a feature which has to be supplied by the agent.

5.1 Application Programming Interface Pro/DEVELOP

Pro/ENGINEER provides the developer with an Application Programming Interface called Pro/DEVELOP. This interface allows external applications to execute operations on Pro/ENGINEER's data (both in memory and on disk) and monitor user functions such as "save a drawing" using pre- and post-handlers. These pre- and post-handlers provide the basis for information flow to an agent. Figure 9 illustrates the interfacing concept in the case of Pro/ENGINEER.

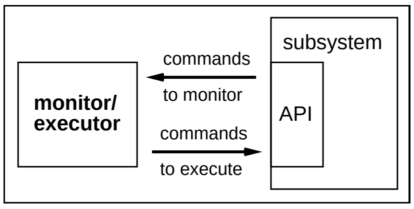

Figure 9: Monitoring and execution by API

Through its application programming interface, Pro/ENGINEER can be enhanced with procedures that are triggered whenever the associated operation is executed. Such a procedure sends a notification message containing the name of the operation to a program which is defined in a Pro/ENGINEER configuration file.

In order that user actions can be monitored, event-handlers have to be installed for all the relevant Pro/ENGINEER operations. An event is the execution of a Pro/ENGINEER operation. It can be fired either before or after the execution of the operation, and, correspondingly, we speak of before-events and after-events. When the event is fired, the corresponding handlers will call procedures running in a separate process using the Pro/DEVELOP library.

The Pro/ENGINEER operations we want to monitor are the so-called dbms-operations. These operations include all the operations for file handling, e.g. the storing of the CAD objects. The label 'dbms' should not mislead: The dbms-operations are just plain file operations and have nothing to do with a DBMS nor do they implement anything of the well-known transaction properties of a DBMS.

The central procedure for the installation of handlers is the Pro/DEVELOP function `prodev_notify`. This function gets a label representing the monitored operation, a switch whether it is the before- or the after-event and the name of the procedure to be called if the event occurs. For example,

```
prodev_notify("menu_dbms_create_object", "begin", NotifyPreInsert);
```

will called the procedure `NotifyPreInsert` before an operation to create an object is executed.

In order to receive acknowledgements that Pro/ENGINEER has completed its task, handlers can be installed which are called *after* the tool's operation has taken place. The form of the installation is analogue to the installation of the before-event handlers.

We have seen in this subsection what possibilities Pro/ENGINEER supplies to monitor its actions. We will now have a closer look at the problems associated with this approach and how we solved them.

5.2 Monitoring module

In theory, it would be very convenient if knowledge about a user action would be obtained before it actually takes place. As we have seen, Pro/ENGINEER supports pre-handlers for the dbms-operations which could do the required task. However, these handlers can only be installed for events fired as a result of the *selection of a menu entry* for a given operation, not the final execution of the requested operation.

This apparently minor difference causes a lot of problems. After the selection of a menu entry, it is not yet clear on which object the requested operation will be performed. Pro/ENGINEER only proposes a default value to the user which is usually the document name of the active window, or, a system-generated name if the object has not yet been saved. This default value can be requested by the before-event handler. It would be possible for that handler to restrict the selection of the object to be saved to exactly the default value. However, this would restrict a user and would change the functionality of the Pro/ENGINEER interface.

Another problem lies in the hierarchical organisation of designs into assemblies and parts. A before-event handler gets only the name of the root assembly if an assembly has to be saved. In order to determine which subordinate assemblies and parts will also be saved, it has to parse the corresponding assembly file and detect which elements have been changed. Since Pro/ENGINEER designs can be very complex, this takes a considerable amount of time. Furthermore, exactly the same action is performed by Pro/ENGINEER itself afterwards, when the real save operation takes place and it passes the names of the saved parts to the after-event handler.

For these reasons, we decided not to use the pre-handlers for the monitoring module. Fortunately, this decision does not impose large problems on the agent because the before-images of the saved objects, i.e. the files with a lower version number attached, are available for a potential compensating operation should the operation have to be undone.

monitored	action	
dbms-operation	pre-handler	post-handler
create, retrieve	—	requests locks, prepare undo
delete	—	buildup delete request
save	—	new object: buildup insert request
		existing object: buildup update request
copy	—	buildup insert request
rename	—	update repository (the local id)
purge	—	maybe save second-newest version

Table 2: Mapping of operations

Table 2 shows how we map the user operations in Pro/ENGINEER to actions in the agent. The mapping of the `purge` and the `save` operations may require further explanation.

`Purge` will delete all old versions of a certain object, i.e. all files with the same name but not having the highest version number. In the case of `screw.part;1`, `screw.part;2` and `screw.part;3`, the operation `purge` consequently would delete `screw.part;1` and `screw.part;2`. As there are cases where `screw.part;2` could be required for a compensating operation, we may first have to save it elsewhere.

In the case of the `save` operation, only one insert or update request is generated in a transaction. This is due to the fact that, for coordination with other tools, only the fact that an update or an insert has occurred is important, and not the possibly numerous intermediate steps created by repeated saves.

5.3 Execution module

We have so far described how operations from Pro/ENGINEER can be monitored and how confirmation is requested for them from the coordination system. In this subsection, we consider the opposite direction concerning requests from the coordination system to execute actions on Pro/ENGINEER data files.

During the development of the execution component, we discovered that very few actions can be performed automatically in theory, and even less in practice. If we regard the three basic global operations insert, delete and update, we can state the following:

insert: Although a CAD design normally cannot be created automatically, one could create an empty file without user interaction. This, however, is not very useful and we therefore did not choose that solution. We decided to inform the user about the necessity to create a new design and leave the rest to him (although certain transaction support is given as we will show later in this subsection).

update: In case of an update request, the situation is quite clear. A user must be involved and no action can be taken automatically by the execution module. User notification is the only possibility.

delete: The deletion of a part or an assembly is the only operation where an automatic approach could be employed. The choice would be to either physically delete the file or move it to another location and inform the user. Unfortunately, there is no Pro/DEVELOP function to remove a part or an assembly (although there is the possibility to monitor the deletion of an object if the user does it interactively). Because of the references of files into other files, it is not possible just to execute a move operation on the operating system level. Therefore, even the delete operation had to be implemented as a request to the user.

There are two ways in which user notification is performed. Firstly, the user who last accessed the relevant object is sent an email message containing the request. Secondly, notification messages are additionally displayed in a special message window attached to Pro/ENGINEER. Of course, a request to the Pro/ENGINEER system is not regarded as being fulfilled simply by sending the notification message. The monitoring module will detect that the requested action has been performed and passes on the information to the main control module.

Consider the case of an `insert` request. The control module checks every insertion message sent by the monitoring module to determine whether it is for an object (id) with an outstanding execution request. If this is the case, the transaction manager in the control module will remove the insertion operation from the tool request and unifies it with the corresponding one from the execution request. Therefore a user can react on an insertion request by saving a new object with the same local identifier as mentioned in the request message. The same mechanism applies for the delete and update operations requested by the coordination system.

In summary, although execution requests sent by the coordination system cannot be performed automatically, we have found an efficient and elegant way to create a semi-automatic solution.

We can contrast Pro/ENGINEER with the classic CAD system, CATIA. CATIA is a monolithic CAD system built by IBM [CAT] and its age is very much reflected in its architecture. Atomicity of operations is not guaranteed at all and the execution of actions by external programs — although it is possible — has to take care of concurrency itself.

In the case of CATIA, the aspect of information flow to an external program is a bit more difficult to classify. Strictly, the operation of a tool should be unaffected by the integration process — at least if no user interaction is necessary for re-establishing global consistency. The user interface consequently has to remain the same. If this condition were to be satisfied, there is no straightforward way for an agent to monitor a user's actions. However, it is possible to extend the CATIA function menus using Application Programming Interfaces (API) such as GII and CATGEO [GII88, CAT92] which allow the call of CATIA internal data management functions. Using these APIs, a completely new function-menu structure can be built which enables the monitoring of all relevant user actions by providing the necessary hooks. But note that existing menus cannot be disabled and therefore bypassing the agent is theoretically possible.

6 Tool Classification

The ease with which an existing tool can be integrated into a distributed information system and the extent to which a full and seamless integration can be achieved depends on the nature of the tool. If the tool is based on an open, extensible architecture and supports distribution standards such as CORBA [OMG91], then the task is reasonably straightforward. However, as discussed, many existing application tools are actually complex systems with a fairly closed architecture and with little or no support to aid the integration process.

We base our classification of tools on three characteristics:

1) The form of communication supported between the tool and an external system.

2) The means by which tool data can be updated.

3) The extent to which the atomicity and durability of updates on data is supported.

Property 1) determines how the agent can obtain information about local operations and further how the agent can influence and control these operations. Property 2) is concerned with how an agent can initiate local operations. Property 3) is concerned with the extent to which the operation of the tool may be considered reliable in terms of ensuring the consistency and currency of its stored data.

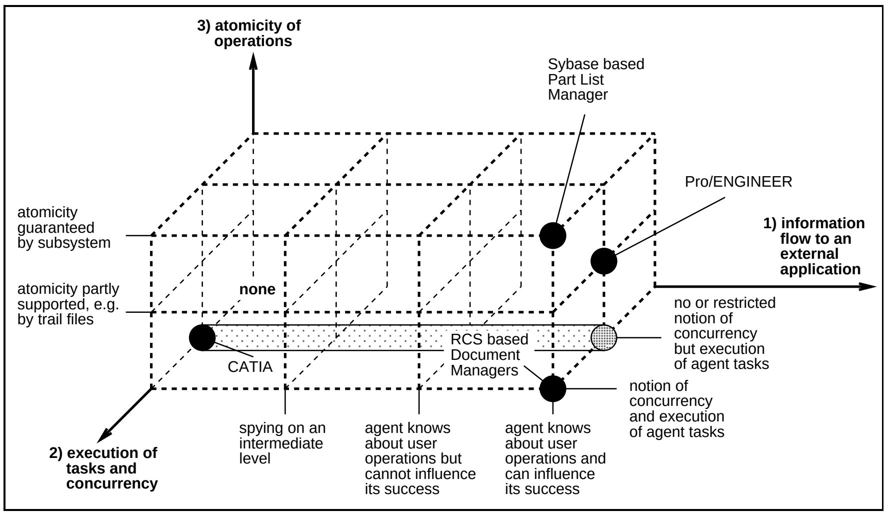
3) atomicity of operations
Sybase based Part List Manager
Pro/ENGINEER
1) information flow to an external application
atomicity guaranteed by subsystem
atomicity partly supported, e.g. by trail files
none
no or restricted notion of concurrency but execution of agent tasks
RCS based Document Managers
CATIA
notion of concurrency and execution of agent tasks
2) execution of tasks and concurrency
spying on an intermediate level
agent knows about user operations but cannot influence its success
agent knows about user operations and can influence its success

Figure 10: Classification scheme for tools

To show the variety of tools and how they may be classified according to these characteristics, we illustrate a classification scheme in figure 10. Each axis corresponds to one of the above characteristics as indicated by the labels. In particular, we show the classification of the RCS and Pro/ENGINEER tools which were described in the previous sections. We will now describe the classification scheme in detail.

6.1 Information flow to an agent

For axis 1), we indicate four possible levels of information flow from the tool to an external application. The worst case scenario is that of a closed tool which provides no information on operations performed and does not permit foreign control over its data and operations. An example of such a tool might be a word-processing system with its data in files. In such a case, the agent might have to resort to monitoring file sizes and modification timestamps combined with tool version dependent knowledge of internal file structures to obtain information about user operations.

At the second level of information flow, we assume that it is possible to obtain information about operations and data at an intermediate level, i.e. at a higher level than sequences of bytes, but a lower level than the tool and user's model of data. An example of such a tool is a CAD system which manipulates geometrical objects such as cylinders but stores the data as sequences of lines, points etc. in a relational database system. In such a case, it may be possible to introduce a layer between the client and server which can monitor and interpret sequences of SQL statements.

A preferred situation is that of being able to obtain information about the tool's operations at the same logical level as that of the user operation. Assuming that information at this level is provided, we can further distinguish the case of obtaining the information *after* the operation has been performed from that where the information is obtained *before* the operation is performed. In the former case, there is no possibility to influence the outcome whereas in the latter case, it is possible to first seek authorisation from the central coordination system before any updates are committed. In the event that a tool supports some form of *undo* actions such that committed actions can be compensated, then the tool can be considered at the preferred level of information flow equivalent to the case where information is obtained about operations before they are committed.

6.2 Execution of external tasks and concurrency

The second aspect used to classify a tool into our framework is important in order that an agent can initiate an operation in its tool in response to a request by the coordination system.

It is preferred that a tool allows external triggering of operations. For example, the automatic creation of part lists, or the setting up of initial frames for technical manuals, are cases where it is desirable that an action can be initiated by an agent. Because a request submitted by the coordination system is not synchronised with any user action, it is important for the construction of an agent whether a tool supports concurrent or quasi concurrent access to its data. If not, the agent has to take care of the concurrency management.

6.3 Atomicity of operations

The third aspect of how a tool can support the construction of an agent and its integration into a distributed information system has to do with system failures. In an ideal world with no system failures, the aspects of information flow to an agent and the triggering of actions, as described in the previous two subsections, would be sufficient to classify a tool. Unfortunately, a tool failure, which results in a loss of data integrity, can occur due to either hardware or software problems. In the case of a stand-alone system, i.e. an application not integrated in a coordinated system, the worst result of a failure would be a full or partial loss of data which, hopefully, would be reduced by restoring data from a backup. However, in the case of a tool integrated in a coordinated system, the state of a tool is at least partially known outside the application, and we must guarantee that all state information made persistent in the coordination system truly reflects the current situation in the tool. This requirement can be fulfilled if the tool guarantees atomicity of operations and data durability together with a way to query its state.

If a tool does not completely provide such a notion, then the agent, with the help of the tool, has to provide it. It may be the case that the tool has some form of log files or trail files, which the agent can use to guarantee durability and atomicity. These files would be used for redoing failed operations or undoing partially performed operations. If the tool does not provide any support, the agent will have to perform all the logging and recovery itself.

In all three degrees of atomicity support described in this subsection, the atomicity related actions of an agent are isolated from the rest of the coordinated system so that the coordination system can deal with a component system (a tool plus its agent) as a database management system. In case of recovery from a crash, however, the coordination system has to be incorporated into the process to obtain necessary knowledge about the state of a component system. Further details on this topic are discussed in [Sch96].

7 Conclusions

We have shown that it is possible to integrate existing tools into distributed information systems by means of agents capable of monitoring and controlling tool operations. While the nature of such agents is tool-specific, it is possible to identify certain classes of tools which determine the appropriate agent architecture. As a result, general guidelines for the development of agents can be produced which, in many cases, facilitate the reuse or adaptation of existing agent modules.

Our experiences in building such agents bring out some interesting points concerning the nature of both the agents and the coordination of tasks in specific application domains.

Contrary to our initial expectations, we could not generally exploit the capability of tools to install pre-handlers for user operations, thereby enabling confirmation from the coordination system to be obtained before execution of the operation. Rather, these pre-handlers were typically used only to prepare a potential compensation of the operation. This was due to the fact that, with these pre-handlers, it is not certain whether the operation will in fact execute and, further, exactly what the operation will be as parameters may be unspecified at this point. For this reason, in both of the agents described in this paper, the transmission of the final user operations was performed in the post-handler part of the monitoring component.

In terms of the nature of task coordination, we found it rare that fully automatic execution of coordination requests is possible, or even desirable. Instead, we discovered that, in most cases, the best solution was some form of semi-automatic execution in which the user receives request notification, either by email or via the message window, and the system then monitors whether the user acts on this request. This approach allows the user to decide what happens to their data, while at the same time automatically detecting that the required task coordination happens.

References

[BD94] P. A. Bernstein and U. Dayal. An Overview of Repository Technology. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Intl. Conf. on Very Large Databases (VLDB'94)*, pages pp 705–713. Morgan Kaufmann, September 1994.

[CAD] Eigner+Partner GmbH, Ruschgraben 133, D-76139 Karlsruhe, Germany. *CADIM/EDB.*

[CAT] IBM Corporation, Marketing Publications, Department 824, 1133 Westchester Av., White Plains, NY 106 USA04,. *The CATIA CAD System.*

[CAT92] IBM Corporation, Department 34CA, Neighbourhood Rd., Kingston, NY 12401, USA. *CATIA Base - Geometry Interface Reference Manual, SH50—91-04*, 1992.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[GII88] IBM Corporation, Marketing Publications, Department 824, 1133 Westchester Av., White Plains, NY 10604, USA. *CATIA - Graphics Interactive Interface (GII) Reference Manual, SH50-0020-0*, 1988.

[NSSW94] M. C. Norrie, W. Schaad, H.-J. Schek, and M. Wunderli. Exploiting Multidatabase Technology for CIM. Technical Report 219, Department of Computer Science, ETH Zurich, July 1994.

[NW95] M. C. Norrie and M. Wunderli. Modelling in Coordination Systems. *Intl. Journal of Cooperative Information Systems*, 4(2 & 3):189–211, 1995.

[NWM+95] M. C. Norrie, M. Wunderli, R. Montau, U. Leonhardt, W. Schaad, and H.-J. schek. Coordination Approaches for CIM. In F. Vernadat, editor, *Integrated Manufacturing Systems Engineering.* Chapman Hall, 1995. (originally appeared in Proc. European Workshop on Integrated Manufacturing Systems Engineering, Grenoble, France, Dec 1994).

[OMG91] OMG. The Common Object Request Broker: Architecture and Specification. Object Management Group and X/Open, Document Number 91.12.1, Rev. 1.1, 1991.

[Par93] Parametric Technology Corporation, 128 Technology Drive, Waltham, MA 02154, USA. *Pro/ENGINEER User Manuals*, 1993.

[Sch96] W. Schaad. *Transaktionsverwaltung in heterogenen, föererten Datenbanksystemen.* PhD thesis, Institute for Information Systems, ETH Zurich,, CH-8092 Zurich, Switzerland, 1996.

[She] Sherpa Corporation, 611 River Oakes Parkway, San Jose CA 95134, USA. *Sherpa: DMS/PIMS.*

[STE92] International Standard Organisation, Technical Committee 184. *ISO DIS 10303, Product Data Representation and Exchange*, 1992.

[VB82] G. M. A. Verheijen and J. Van Bekkum. Niam : An information analysis method. In T. W. Olle, H. G. Sol, and A. A. Verrijn-Stuart, editors, *Information Systems Design Methodology*, pages 537–589. North-Holland, 1982.