An Architecture for Multi-User Software Development Environments

Israel Z. Ben-Shaul Gail E. Kaiser George T. Heineman

Department of Computer Science
Columbia University, New York, NY 10027

We present an architecture for multi-user software development environments, covering general, process-centered and rule-based MUSDEs. Our architecture is founded on componentization, with particular concern for the capability to replace the synchronization component – to allow experimentation with novel concurrency control mechanisms – with minimal effects on other components while still supporting integration. The architecture has been implemented for the MARVEL *SDE.*

1 Introduction

Software Development Environments (SDEs) emerged in an attempt to address the problems associated with developing, maintaining and managing large scale software projects. One of the main issues in SDE research is how to construct environments that are integrated, while at the same time flexible and extensible. Although there have been numerous proposals for cooperative transaction models [4], little has been achieved regarding flexibility and extensibility of such *synchronization* mechanisms for multi-user SDEs from the system-architecture point of view. Throughout the paper we refer to this aspect of an SDE as the "multi-user" property.

The architectures of process-centered SDEs include process enaction engines, which enable a programmable approach to defining the behavior of an environment to support a particular software development process [22]. The process enaction engine and the corresponding process modeling language must be extended with a notion of concurrency consistency and corresponding synchronization primitives to support multi-user environments, where the process as well as the data is shared.

In many process-centered SDEs, the process is defined in terms of rules and enaction is achieved through rule chaining. Examples include CLF [24], Oikos [1] and Merlin [30]. Such SDEs must support synchronization among automated chains of activities as well as activities directly invoked by users. In any multi-user SDEs, the architecture must also support interprocess communication, scheduling and context switching, transaction and lock management, and other facilities on which synchronization depends.

This paper presents an architecture for multi-user SDEs (MUSDEs) that is intended to support the requirements of general, process-centered and rule-based MUSDEs. The emphasis is on identification of the system's components and on the interfaces and interrelations among them rather than on application of specific synchronization policies. We have implemented the architecture for MARVEL, which was previously a single-user system [20]. This work is complementary but orthogonal to the research done by Barghouti and Kaiser on cooperative transaction management for SDEs in general and MARVEL in particular. The focus of their work has been on *modeling* coordination and cooperation, whereas here we focus on the *architectural* facilities that enable the implementation of such sophisticated synchronization mechanisms.

In section 2, we give the necessary requirements that an MUSDE must fulfill, by definition, and additional desired properties. Section 3 introduces the architecture, its main characteristics and functionality. Section 4 explains the rationale behind the architecture. Section 5 describes the implementation for MARVEL and our experience, including changing and tailoring some of the components. Section 6 compares to related work. Section 7 briefly evaluates the architecture and summarizes our contributions.

2 Requirements

Data-sharing - We distinguish between "product" data and "control" data: the former represents the actual data elements under development (i.e., source files, object files, design documents, etc.), while the latter represents the data used by the SDE to manage the project. Examples of control data for a source file include its version, compilation status, reservation status, etc. Product data may be integrated with con-

trol data (e.g., an object is defined as having "control" state attributes and file attributes that point to "product" items) or may be maintained separately. In general SDEs, control data represents the status with respect to a hard-coded policy, whereas in process-centered SDEs, control data reflects the state of the specific process in action.

Data-consistency - An MUSDE synchronizes concurrent access to the SDE's data to maintain its consistency, e.g., it prevents data from being garbled by conflicting accesses (such as multiple independent updates) to the same or related data items. Product data can be maintained either by the SDE or in the file system; however, control data must be maintained by the SDE. But access to both must be synchronized, either in a centralized or a distributed fashion, and in the latter case can be tightly integrated within each user's workspace or separate in a DBMS.

Process-sharing and process-consistency - In addition to *data*-consistency as above, which is required for all MUSDEs, process-centered SDEs must maintain *process*-consistency, as specified in the process modeling language. Thus, the process engine must maintain a global view of the process. Again, this can be done in either a centralized fashion, or in each user's workspace provided that the necessary information is replicated among users. For example, consider a constraint taken from the "ISPW problem" [13], where a member of group PROGRAMMER cannot make any code changes before some or all members of the Configuration Control Board have given approval. The MUSDE must ensure that the constraint is applied to all involved participants (or at least programmers and CCB members).

Whereas the above characteristics are *required* in MUSDEs of the indicated classes – by definition – the following represent additional properties desired in an MUSDE. These properties together form the basis for the rationale behind our architecture.

Perhaps the most important property from the architectural point of view is *flexibility in selection and application of synchronization mechanisms*. The idea is to be able to replace or modify concurrency control policies, both globally (i.e., for all users of the system) and locally (among selected groups of users). Some proposed concurrency control models, such as transaction groups [15], support this capability to a limited extent in that the policy for each group can be specified in a formalism supplied by the implementation. What we have in mind is more general: The architecture should be constructed such that the entire synchronization component can be replaced with minimal (preferably no) code changes to other parts of the system. This enables to conduct cost-effective experimentation, which is important in such a novel research area.

The architecture should support synchronization components whose transaction models range from classical atomicity and serializability to *long, interactive operations* and *cooperation*. Any synchronization mechanism for an MUSDE must take into account that many activities in software development are long – conventional atomic and serializable transactions are not suitable, and interactive – response time is more important than overall throughput. Cooperation is needed to enable sharing and exchange of information during parallel development.

Extensibility and broad scope of application - An MUSDE should be able to be extended with new tools, including tools not specifically developed for the MUSDE.

High Visibility - An MUSDE implemented on a window-based platform should provide users with graphical visualization of at least the control data, and preferably the product data as well. Since SDEs often support complex and highly structured data models, it is especially desirable to be able to display the types and relationships of all objects of the environment. This means that an MUSDE has to maintain up-to-date information as it is dynamically changed by multiple users.

Recovery - Persistence of product data can be provided by the host file system, but persistence of control data must be provided by the MUSDE. Recovery is an important aspect that ensures consistency of persistent data in case of external and internal failures. We distinguish between concurrency control, which is required by definition, and recovery, which – although not mandatory – is a highly desired property for industrial-strength environments. Traditionally, these two functions are both carried out by the "transaction manager".

3 The Architecture

Two major principles underlie the overall design: *componentization* and *layering*. According to the componentization principle, a complex system should be built from independent, loosely-coupled and replaceable components. These components must have flexible interfaces and support a variety of different policies potentially employed by alternative interacting components (i.e., components that provide the same services in different ways). Layering is a paradigm in which each component provides services only to the next higher layer and receives services only from lower levels. Layering lowers complexity by reducing inter-component linkages.

Componentization is becoming popular in operating systems (e.g., the replaceable pager in Mach) and databases [25], and the layering approach has been followed in many areas such as communication protocols [9] and databases [6]. The combination seems especially promising for SDE technology, which is by nature subject to changes [28]. We suggest the potential to revise any system component (although with differing degrees of difficulty). Our major concern here is to be able to modify the synchronization mechanism with minimal effects on task management and the re-

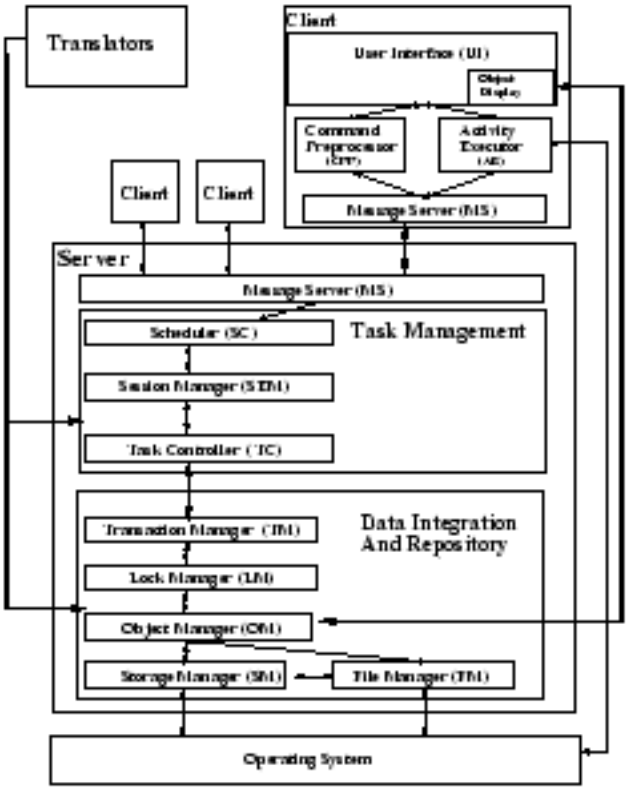

Figure 1: An Architecture for Multi-User SDEs

mainder of data integration and repository management, which implies that transaction and object management should be separate (as in Camelot [12]).

The generic architecture is depicted in Figure 1, using the terminology of the "toaster" reference model [11]. We concentrate here on explaining *how* things work, and defer to section 4 the discussion on *why* we chose to design the architecture this way.

The architecture follows the conventional client-server model. Each active environment with a populated objectbase is managed by a single centralized server, and multiple clients are distributed, each representing a user *session* that lasts from invocation to exit. Both the server and each client are implemented as individual operating system processes. Each client serves as a front-end to the human user and as an activity execution unit. The concept of *activity* encompasses all operations done to product data, such as editing and testing, via internal or external tools; it does not include operations on control data (although these might occur as side effects). Clients may spawn child processes to execute activities.

The server provides *data integration and repository* and *task management* services. Service requests always originate at a client, but most requests are sent to the server after client preprocessing. The server validates and processes the request before returning to the client with the desired information and/or instructions to the client to execute a specific activity. The server mediates access to both control and product data, and modifies control data according to the environment specifications.

The architecture distinguishes between normal users and an *environment administrator*. The administrator's role resembles that of the Data Base Administrator in DBMSs. The administrator uses a privileged client to define the data model (schema) and any integrity constraints on the data; the process model, if any; and the programmable aspects of the synchronization policy, if any.

3.1 Task Management

Translators - Any SDE that allows to define the data model for the control and/or product data must have a data definition translator. In process-centered environments, a process model translator is also needed. MUSDEs with programmable synchronization require yet another translator. Translation can be on-line, in which case this component acts as a loader for other interpretive components, or off-line, in which case it "compiles" the specifications into internal form and is not actively involved at run-time.

Scheduler (SC) - Schedules requests from clients for services, including context-switching. Before a client is serviced, SC makes two contexts active: the client's session-context and the specific task-context.

Session Manager (SEM) - Encapsulates an entire session between a specific client and the server, that is, all requests that occur from invocation to exit of the client. SEM can: (1) maintain the user-specific environment and operating system parameters for general configuration purposes; and (2) store enforcement information that pertains to the entire session (as opposed to task-specific information). For example, users might explicitly "attach" to a specific subprocess to perform during that session.

Task Controller (TC) - This is the central component of the environment, which provides most of the services to the client. A *task* is defined as any activity initiated by a client together with all the derived operations carried out by the environment, such as automation and enforcement actions. For example, an SDE might have a constraint that when an interface to a function **F** is modified, all source files that call **F** must be marked for modification. The modification of **F** and the marking of dependent files is considered one task. In a general MUSDE, TC may degenerate to a command interpreter, perhaps with a query processor. In process-centered environments, this component includes the process engine, in charge of enacting the process. TC operates in the context of the current session, but maintains a task context for each active task in the system.

3.2 Data Management

Transaction Manager (TM) - Maintains the integrity of the data in case of concurrent access and

failures. In process-centered MUSDEs, TM also maintains process-consistency. However, it is not responsible for *detecting* any conflicts due to concurrent access, but only for *resolving* them.

A "transaction" can map to a single activity or to a single task, but usually not to a session, since this would imply coarse-grained concurrency. There are no specific guidelines for the implementation of concurrency control or recovery, except for the restriction to locking-based mechanisms. For example, an environment may use a "blocking with deadlock resolution" mechanism or a "non-blocking with abort" mechanism, or a combination of both. Also, TM may use flat transactions or nested transactions that model the nesting of activities and subtasks within a task.

TM-TC interface - The interface between TC and TM is a critical issue as it bridges between the task level and the data level. It is desirable for TM to be independent of any specific task model and for TC to be independent of any specific transaction model, so that either can be replaced with minimal overhead. A predefined set of transaction primitives known to TC must be supported by any TM, with the set flexible enough to support many synchronization policies. However, semantics-based concurrency control inherently requires some knowledge of the task level to resolve conflicts that are context-sensitive. This implies that the TC-TM interface may need to be augmented with a mediator component that reconciles information from both levels.

Lock Manager (LM) - Usually considered a subcomponent of the transaction manager, LM is treated in our architecture as a separate component. Its main role is to detect any potential violations of the data-consistency constraints, as defined by a lock-compatibility matrix. LM must permit a broad range of lock modes to enable TM the freedom to choose those that meet its needs. But the separation of LM, TM and OM makes it impossible to predict what lock set and compatibility will be needed. However, viewing LM merely as a "mechanical" conflict detector enables it to be table-driven, with the tables loaded during system initialization. This means customizations of TM affect LM only through the tables.

An additional property of LM is to be able to hold *multiple* locks on objects, on insistence from TM, even when they violate the defined compatibility. This is useful for implementation of non-conventional concurrency control policies. For example, transaction groups may allow several transactions in the same group to share transient results. ObServer [26] is a multi-user data server with a rich lock set, including communication modes (for notification), which is capable of supporting transaction groups. This approach provides flexibility in transaction management but is not extensible. In contrast, we regard lock management as a mechanism to detect conflicts only, for an arbitrary set of lock modes; ObServer's communication modes can be implemented in LM with proper support from TM as part of conflict resolution.

Object Manager (OM) - Implements the data model, provides persistence, and performs all requests for access and modification of both control and product data. We assume a generic object-based data model with optional class ("is-a") hierarchy, composition ("is-part-of") hierarchy, and arbitrary relationships between objects ("links"). An object may represent purely control data, an encapsulation of product data, or a combination.

OM-LM interface - For componentization to work, it is important that OM provide the upper layers with an object abstraction that avoids concern with internal representation. Further, upper layers should not know whether objects are in main or secondary memory. The main difficulty with separating OM and LM is that data-consistency specifications may need to be extended for a specific data model. For example, composite objects and links among objects may require "intention" lock modes for ancestor and linked objects, respectively. This predefined set of lock "extensions" is understood by LM, allowing a wide variety of object-based data models but precluding the possibility of replacing objects with a radically different form such as relations (such a change would also seriously affect the upper layers, notably query processing in TC, impeding componentization).

Storage and File Manager (SM and FM) - SM is responsible for low-level disk and buffer management for control data. It manages untyped, raw data, and interacts with the underlying operating system. If the MUSDE uses file-based tools and maintains its product data in ordinary files, FM is responsible for accessing the files requested by OM (in a shared file system such as NFS only path names need be passed). When product data is encapsulated within control data, objects usually abstract the file system by providing typing and relationship information. In this case SM is responsible for both, and FM degenerates into a mapping function between objects and files.

The separation between storage, object, lock and transaction management distinguishes our architecture from most other systems that provide data management.

3.3 The Client

User Interface and Objectbase Display Manager (UI) - Provides the human user interface to all environment services, including a display of the *entire* objectbase structure (subsets can be viewed via browsing). This feature introduces the challenge of keeping the display up-to-date, since the objectbase is dynamically changed by multiple users, including modifying, adding and deleting objects and/or relationships between objects.

Activity Executor (AE) - Interacts with tools in an environment-specific manner. This might include interaction with the operating system for spawning child processes with suitable command lines and transforming data to/from objectbase and tool formats. There

may or may not be communication between the AE and the activity and between the AE and the server while activity execution in in progress.

Command Preprocessor (CPP) - This component is open-ended. It includes formatting of requests for services so that they conform to the interface specifications of the various service providers in the server (fronted by TC), and executes local services that do not affect other users or the software development process. An example of the former is an ad-hoc query parser that performs syntax checking and passes to the server a parsed query. An example of the latter is the "help" facility. CPP has no significant impact on the overall architecture.

Message Server (MS) - Transfers information between the client and the server over the communication medium. MS must preserve the object abstraction so that both ends can refer to objects identically, which means it must provide linearization and delinearization of the objectbase structure.

Mapping our architecture to the "toaster" model, data integration and repository management services are in the server, and user interface services are in the clients, as expected. The interesting mapping is that of task management. We divide this between the client and the server, where the client is responsible for "activity execution" and the server for the rest.

4 Decisions and Justifications

4.1 Client-Server Separation

The first issue to consider is the degree of distribution. The two obvious alternatives are to fully centralize services or to fully distribute them among clients. In the first case there would still be minimal clients, at least operating system shells, to allow multiple users to communicate with the environment; but all control and product operations would take place in the server. In the second case there would be no dedicated server at all, but only clients, with all control and product operations executed in a client and shared only via communication directly among clients.

We chose a hybrid approach, in which clients are responsible for long duration activities and the server is responsible for relatively short term task control and synchronization. Maintaining data- and process-consistency internal to the server reduces communication overhead, while farming out interactive and/or computation-intensive activities to the relevant clients keeps computation overhead low and response time high. This division of labor seems to best exploit today's high performance workstations and high capacity server machines.

Locating task control in the server does not preclude the possibility of different "views" for different clients: they can be managed by the server as part of the session context. Further, the server-client separation does not prevent distribution of the server itself into multiple server processes, with communication among themselves to handle decentralized data, process and synchronization. Our intent is to instead make an inherent distinction between the *roles* of clients and server(s).

4.2 Transaction and Lock Management

The main reason for decoupling TM and LM is to distinguish conflict detection from conflict resolution, where the former is a mechanical procedure that reports any violations of the defined consistency and the latter is an elaborate procedure that decides how to resolve a conflict when it arises. This separation enables to modify and/or replace synchronization policies without affecting the underlying conflict detection. Furthermore, the fact that LM has no knowledge of the semantics of the various lock modes enables to implement LM in a way that it can be reconfigured externally via tables, without any code changes. The decoupling of transaction management from lower levels also brings TM closer to task management, enabling semantic-based concurrency-control without concern for low-level data management. This separation contributes perhaps more then anything else to the flexibility of the system with respect to concurrency control.

4.3 Tunable Lock Management

The alternatives are: (1) a non-locking policy, where concurrency control is optimistic (as in NSE [18]); (2) a hard-coded lock set and lock-compatibility matrix; and (3) a dynamic lock set and lock-compatibility matrix. We addressed hard-coding versus externally-defined lock tables in section 3.2. Optimistic concurrency control may be useful when conflicts are known to be rare, provided that the "resolution" is done by "merging" changes from conflicting operations, since rolling back long and/or interactive operations would be unacceptable in most situations. However, an effective merging procedure for source code is still beyond the state of the art (as evidenced by [19]), and there is no general way to merge two versions of a data file created by conflicting operations (although [14] gives some hope of advances).

4.4 Objectbase Visibility

The two obvious alternatives are to keep an entire replica of the objectbase at each client, or to display only those objects that are actually used by a client. Note that in any case control data is manipulated in the server, so the issue is not where to modify the data, but rather how to display it. The main problem with keeping entire replicas is that it is expensive and unnecessary, since objects in a MUSDE can be very large and may change frequently, causing tremendous communication overhead. On the other hand, displaying

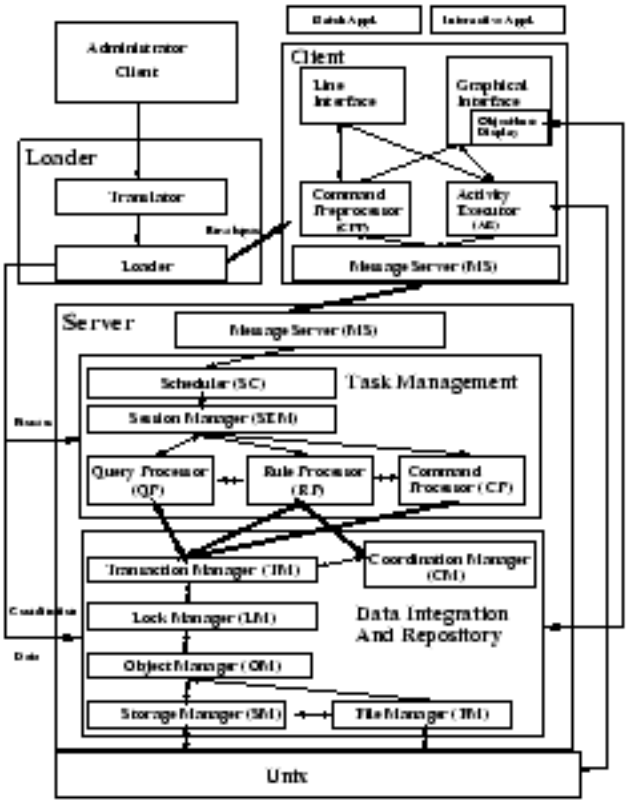

Figure 2: MARVEL 3.x Architecture

only objects in current use does not fulfill the "high visibility" property.

We chose an intermediate approach, in which the structure of the objectbase is maintained, but not its contents. For each object, we maintain a cache of its name, type, unique ID and relationships to other objects. This provides sufficient information for viewing the entire objectbase, while still compact in volume for transmission by MS.

Another consideration is the display-refresh policy. The alternatives are to: (1) broadcast every change to all active clients; (2) refresh periodically; and (3) refresh "on demand", as determined by the server, by "piggybacking" the refreshed image on the next message sent to the client. The third alternative is preferred as it saves communication overhead while keeping information reasonably up to date.

5 Implementation for Marvel

The MARVEL 3.x architecture is illustrated in Figure 2. It can be viewed as a rule-based instance of the generic MUSDE architecture of Figure 1. The client structure is essentially the same. The server reflects TC in three sub-components: query processor (QP), command processor for built-in commands (CP), and rule processor (RP) responsible for process enaction. It also adds a Coordination Manager (CM) as a mediator between TM and TC. MARVEL's translation and loading component is collectively called Loader. Tool envelopes and data, process and coordination models are written (by the administrator) in various notations and loaded (again by the administrator) using a privileged client, tailoring the environment's behavior according to these specifications. The MARVEL daemon, not shown, automatically starts a server on the appropriate objectbase when its first client logs in, and shuts down the server after the last client has exited.

5.1 Process Modeling and Enaction

The process is defined in terms of rules, each representing a single activity. Each rule consists of a *name*; a list of typed *parameters*; a *condition* that represents the properties that must hold on actual parameters and other objects bound in the condition for the rule to fire; an *activity* that specifies a "product" activity and its arguments; and a set of mutually exclusive *effects* consisting of assertions to the objectbase that reflect the possible results of executing the activity. Rules are implicitly related to each other through matches between a predicate in the condition of one rule and an assertion in the effect of another rule.

Process enaction in RP is done through *chaining*. When an activity is requested, the condition of the corresponding rule is evaluated. If not satisfied, RP attempts to satisfy it by backward chaining to other rules whose effects may satisfy the user-invoked rule. This is done recursively, until the condition is satisfied or all possibilities are exhausted, in which case the activity cannot be executed. When the activity returns from the client (assuming the rule's condition was satisfied), RP asserts the effect indicated by the status code returned from AE and then recursively forwards chains to all rules whose conditions have become satisfied. MARVEL distinguishes between *consistency* and *automation* chains, which are specified by annotations on condition predicates and effect assertions in the rules [5]. Consistency chains propagate changes and are by definition mandatory and atomic. Automation chains automate activities and are by definition optional; they may be terminated at any point or "turned off" entirely.

5.2 Task Management

MARVEL's scheduler implements a simple FCFS non-preemptive scheduling policy. However, non-preemptive scheduling does *not* imply that an entire session, or even an entire task, is handled by the server atomically. Instead, we exploit the natural "breaks" within and among tasks, at which points the server performs a context switch and turns to the next client request. That request might resume an in-progress task or initiate a new task.

RP is the heart of task management. A task consists of all rules executed during backward chaining,

followed by the user-invoked rule (which caused the backward chain), followed by all rules executed during forward chaining. RP operates in a specific task context, consisting of information necessary for maintaining the state of the task. The main data structure is the *rule stack*, one per task. Since backward chaining is multiply-recursive and generates an AND/OR tree (i.e., in some cases a rule's condition may be satisfiable only by application of a set of rules, and in other cases by any one of many possible rules), the rule stack is implemented as a multi-level stack, where each level consists of an ordered set of rules that correspond only to the first rule in the previous level, and are not related to other rules in the previous level. The same stack is used for forward chaining, although here a standard stack mechanism is sufficient.

One problem of multi-tasking rule processing is that multiple instances of the same rule may be fired concurrently by the same or different clients, and since they all fire in the context of one RP (i.e., one address space), rules cannot contain any private data. This problem is solved by making rules *reentrant*. Each invocation entails creation of a rule-frame, which consists of a pointer to the (read-only) rule and a dynamically allocated data section, which it retains throughout the entire life cycle of a rule chain.

5.3 Data Management

TM supports a nested transaction model in which a task is modeled as a top-level transaction, each consistency chain is a subtransaction consisting of a further level of subtransactions corresponding to individual rules, and each rule in an automation chain is an independent subtransaction on its own. By definition, an entire consistency chain is executed to completion or rolled back as if it never started, while the latest rule in an automation chain can be aborted without affecting the rest of the chain. In MARVEL 3.1, CM will serve as a mediator between data and task management; CM-RP and CM-TM interfaces have already been partially implemented.

MARVEL's composition hierarchy is based on ORION [21], using intention locks for ancestors. When object **O** is locked, all **O**'s ancestors are locked in the corresponding intention mode. Intention locks are generally weaker than the corresponding descendant locks, and their goal is to protect objects from being affected by an operation on an ancestor. For example, when object **O** is locked in **L** mode, **IL** locks are placed on all its ancestors, where **IL** is compatible with any operation that would not affect **O**. In particular, it is compatible with another **IL** lock. This idea can be extended to linked objects as well as ancestors, but this is not supported in MARVEL.

LM reads three tables when initialized: *compatibility matrix*, *ancestor table* and *power matrix*. The compatibility matrix defines the set of lock modes and the compatibility of any two lock modes. The ancestor table indicates which lock to apply to the ancestors of the object being locked in a certain mode. The power matrix determines which lock has precedence given two locks requested by the same transaction.

SM uses the Unix dbm package. Although more sophisticated data management strategies can be supported by dbm, SM loads the entire objectbase (but not files) into memory at server startup. FM is implemented by a collection of system calls that map the object name-space to the file system name-space, and perform operations on a "hidden" file system rooted at a directory representing the populated objectbase of an environment. MS uses Internet sockets.

5.4 Client and Loader

UI includes both graphical and command line interfaces with the former providing full objectbase browsing capability (conceptually communicating with OM directly) and the latter supporting batch processing scripts as well as dumb terminals. CPP includes an ad-hoc query parser. AE is the most complex component of the client. It is in charge of spawning child processes for executing *envelopes*, basically shell scripts, for the various tools defined in the environment. AE communicates with envelopes through pipes in a "black-box" fashion: inputs are provided at the beginning of activity execution, and output and a status code are collected at the end [16].

The Loader generates a static rule network from the process model, which is used at runtime to determine chaining. This network is loaded into RP, to define process-consistency. The data and process models are tied in the sense that rule parameters and local bindings in the conditions of rules are typed according to classes. The data model is used by OM and QP. The various lock tables (a degenerate coordination model) are loaded into LM, to specify data-consistency. A full coordination model using the Control Rule Language syntax described by Barghouti [3] can already be loaded, but is not included in the current release.

5.5 Experience

We started with the standard shared and exclusive locks, and intention locks (Figure 3-a), but then modified the lock tables several times. The final change added two new lock modes and removed one, and changed the compatibility of old modes. The purpose was to provide semantics-based locking by distinguishing between operations that affect only a single object (e.g., write on a simple attribute) and operations that might affect related objects (e.g., the delete operation removes an object and all its children). Strong Exclusive (SX) and Strong Shared (SS) locks were added, and X and S became compatible with any intention lock (see Figure 3-b). The ancestor table was modified to include intention locks for the new modes. This required no code changes to LM and only minor changes to TM to replace requests for locks according to the new semantics. Even these code changes would not

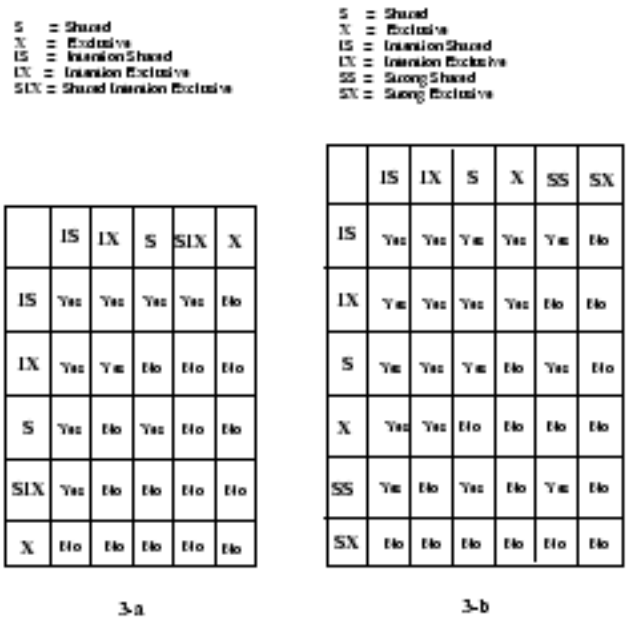

Figure 3: Initial and Revised Compatibility Matrices

have been needed for MARVEL 3.1, where the specific lock modes to use for particular arguments of activities and rules will be specified in external tables, as part of the coordination model. Thus, a dramatic change in conflict detection can be achieved with very small overhead.

We also started with a flat transaction model, in which an entire chain executed as a single transaction. This made it impossible to treat different subsets of a task differently. For example, we could not abort an automation subchain without rolling back consistency subchains descended from the same user-invoked rule. We replaced TM with nested transactions. Each rule triggered during an automation chain, together with any consistency subchains emanating from it, is a subtransaction that can be aborted without affecting the top-level transaction or other subtransactions. Again, this major change had no impact whatsoever on LM, and required only trivial changes to RP.

CM is being developed to support programmable conflict resolution for MARVEL 3.1. The administrator will be able to define an optional set of *control rules* to specify scenarios when the default policy above may be relaxed, and prescribe appropriate actions in each such case. We have already built a facility whereby CM accesses RP's rule stacks, since control rule scenarios require inspection of conflicting rule chains. TM already requests CM to try to match its control rules, but we are still investigating the desirable operations for the actions, so the semantics of a non-empty control rule base are undefined. We will have to add to TM a "marking" phase, to annotate objects left temporarily inconsistent by control rule actions that suspend or terminate in-progress consistency chains, and an "unmarking" phase to attempt to restore consistency when access to such objects is requested. We anticipate no other changes to TM, small if any to RP, and none to other components.

5.6 Status

MARVEL is implemented in C and runs on SparcStations (SunOS 4.1.1) and DecStations (Ultrix 4.2), using X11R4 Windows. MARVEL 3.0 and 3.0.1 have been licensed to about 25 educational institutions and industrial sponsors since December 1991. 3.0.1 includes all the features presented in this paper, except where noted, and was the first version fully developed and maintained using C/MARVEL, a MARVEL process for team programming in C. We are currently using C/MARVEL to develop MARVEL 3.1, planned for release in 1993. In addition to the enhancements explained above, 3.1 will include: An XView user interface, limited data and process evolution for existing objectbases, and integration of built-in commands with the rule chaining engine.

6 Related Work

Gypsy (aka SMS) [8] is an extended version control system that is tightly integrated with an extended operating system. Synchronization is manual, and users work in isolation, each in his/her own "workspace". Although Gypsy provides a mechanism for multiple users to access data objects concurrently by specifying a list of users that can attach to a workspace, it provides no means for coordinating their access.

Arcadia [29, 23] is a process-programming environment based on research in SDE technology underway by the Arcadia consortium. Like our architecture, it is constructed out of layered components that are intended to be replaceable. However, although process-consistency from the process-programming point of view is addressed extensively by Sutton [27], an independent synchronization component is conspicuously absent from the architecture. We guess that multi-user synchronization is provided by the object management system.

Melmac [10] is a process-centered environment with a client-server architecture, in which the server is primarily concerned with data management and provides a simple transaction mechanism, and the clients are responsible for process enaction. One shortcoming evidenced by the examples given in [17] is that since process management is detached from the server, it seems that rule chains cannot be interleaved even during activity execution, which might degrade response time significantly.

Oikos is a rule-based MUSDE that supports concurrency using a hierarchy of blackboards that resemble Linda's tuple spaces. Oikos enables to specify a wide range of services as part of process enactment, including database schemas and transactions. However, while concurrency is an inherent aspect in the

Oikos architecture, concurrency control is not, and it is not clear what range of synchronization policies can be supported, nor how these might be supported.

CLF is a rule-based MUSDE that distinguishes between consistency and automation, but through separate classes of rules rather than annotations on rule predicates as in MARVEL. CLF employs a form of optimistic concurrency control based on merging, with inconsistency *tolerated* by automatically placing guards on inconsistent data [2], similar to our notion of "marking". Changes are grouped into evolution steps, which can be undone or redone [7].

Merlin is the closest system to MARVEL. From the process modeling viewpoint, the main difference may be that Merlin distinguishes forward and backward chaining styles of rules while MARVEL has a single rule base and a symmetric chaining model. There are substantial architectural differences, however: Merlin employs a simple checkin/checkout model, using an object's state as determined by the rules as a lock; there is no support for multiple locking modes; and the objectbase display is limited to each user's working context (although there is a refresh mechanism). It appears that chaining operates in each user's working context (client) as opposed to a centralized server.

7 Evaluation and Contributions

Semantics-based concurrency control and componentization are, in some sense, conflicting goals: how can the transaction manager be semantics-based when the semantics are hidden in the task controller? For example, in our work towards programmable concurrency control, we will have to develop a richer interface between the rule processor and the coordination manager than was previously needed for the transaction manager. It seems unlikely that a sufficiently rich general interface – without a sophisticated mediator – can be developed between the task controller and the transaction manager to allow replacement of either without affecting the other.

Our architecture provides no direct interface between clients and the synchronization components. However, users will need to place explicit requests for notification, if not other purposes; we anticipate changes would be required for the command preprocessor as well as the coordination and/or transaction managers.

The most significant drawback of our architecture is that the single centralized server does not scale up to very large numbers of clients. As more clients are added and the objectbase grows, the likelihood of noticeable waits increases. This is an important area for future research.

But there are many advantages of our architecture. At the user interface level, the structural display facility provides for high visibility without the overhead of maintaining complete replica at the clients. At the task management level, the separation between activity execution and task control provides for process sharing while enabling local execution of tools.

At the data management level, we have made several architectural decisions we believe are unique as well as fruitful: (1) A table-driven lock manager allows to modify data-consistency policies with no code changes. (2) The separation between transaction and lock management allows definition and monitoring of data-consistency independent of the synchronization policy, with minimal overhead. Moreover, this enables to implement sophisticated coordination models, with little effect on other components. (3) The decision to separate transaction management from object management emphasizes our view of support for advanced synchronization models. Essentially, we moved synchronization away from low-level data integration and closer to the semantic, task level. We do not know of any other MUSDE with such functionalities.

Acknowledgments

We would like to thank Naser Barghouti for his numerous contributions to the MARVEL project; conversations with Bill Riddle, Brian Nejmeh and Steve Gaede helped shape the functionality of multi-user MARVEL; Mark Gisi, John Hinsdale, Tim Jones, Will Marrero, Moshe Shapiro and Mike Sokolsky participated in the implementation effort; Hideyuki Miki, Steve Popovich and students in the E6123 Programming Environments and Software Tools course helped by testing MARVEL as users.

References

[1] V. Ambriola, P. Ciancarini, and C. Montangero. Software process enactment in Oikos. In *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 183–192, Irvine CA, December 1990.

[2] Robert Balzer. Tolerating inconsistency. In *13th International Conference on Software Engineering*, pages 158–165, Austin TX, May 1991.

[3] Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University, February 1992.

[4] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[5] Naser S. Barghouti and Gail E. Kaiser. Scaling up rule-based development environments. In *3rd European Software Engineering Conference*, pages 380–395, Milano, Italy, October 1991.

[6] Alfred Brown and John Rosenberg. Persistent object stores: An implementation technique. In *Implementing Persistent Object Bases: Principles and Practice*,

pages 199–212, Martha's Vineyard MA, September 1990. Morgan Kaufmann.

[7] Don Cohen and K. Narayanaswamy. A logical framework for cooperative software development. In *6th International Software Process Workshop*, Hakodate, Japan, October 1990.

[8] Ellis S. Cohen, Dilip A. Soni, Raimund Gluecker, William M. Hasling, Robert W. Schwanke, and Michael E. Wagner. Version management in Gypsy. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 201–215, Boston MA, November 1988.

[9] J. D. Day and H. Zimmermann. The OSI reference model. In *IEEE*, volume 71, pages 1334–1340, December 1983.

[10] Wolfgang Deiters and Volker Gruhn. Managing software processes in the environment MELMAC. In *SIGPLAN '90 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 193–205, Irvine CA, December 1990.

[11] Anthony Earl. Principles of a reference model for computer aided software engineering environments. In *Software Engineering Environments International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 115–129, Chinon, France, September 1989. Springer-Verlag.

[12] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon A Distributed-Transaction Facility*. Morgan Kaufman, 1991.

[13] Marc I. Kellner *et al.* Software process modeling example problem. In *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 176–186, Redondo Beach CA, October 1991.

[14] Martin Hardwick *et al.* Change processing tools for concurrent engineering. Technical Report 91-5, Rensselaer Design Research Center, Rensselaer Polytechnic Institute, February 1991.

[15] Mary F. Fernandez and Stanley B. Zdonik. Transaction groups: A model for controlling cooperative work. In *3rd International Workshop on Persistent Object Systems: Their Design, Implementation and Use*, pages 128–138, Queensland, Australia, January 1989.

[16] Mark A. Gisi and Gail E. Kaiser. Extending a tool integration language. In *1st International Conference on the Software Process*, pages 218–227, Redondo Beach CA, October 1991.

[17] Volker Gruhn. *Validation and Verification of Software Process Models*. PhD thesis, Forschungsberichte des Fachbereichs Informatik der Universität Dortmund, 1991.

[18] M. Honda. Support for parallel development in the sun network software environment. In *2nd International Workshop on Computer-Aided Software Engineering*, pages 5–5 – 5–7, 1988.

[19] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 234–245, White Plains NY, June 1990.

[20] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[21] Won Kim, Jorge F. Garza, Nathaniel Ballou, and Darrel Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[22] Leon Osterweil. Software processes are software too. In *9th International Conference on Software Engineering*, pages 1–13, Monterey CA, March 1987.

[23] Leon J. Osterweil and Richard N. Taylor. The architecture of the Arcadia-1 process centered software environment. In *6th International Software Process Workshop*, Hakodate, Japan, October 1990.

[24] CLF Project. *CLF Manual*. USC Information Sciences Institute, January 1988.

[25] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in EXODUS. In *SIGMOD International Conference on the Management of Data*, pages 208–219, San Francisco, CA, May 1987.

[26] Andrea H. Skarra, Stanley B. Zdonik, and Stephen P. Reiss. An object server for an object-oriented database system. In *1986 International Workshop on Object-Oriented Database Systems*, pages 196–204, Pacific Grove CA, September 1986.

[27] Stanley M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming*. PhD thesis, University of Colorado, 1990.

[28] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. Language constructs for managing change in process-centered environments. In *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 206–217, Irvine CA, December 1990.

[29] Richard N. Taylor, Richard W. Selby, Michal Young, Frank C. Belz, Lori A. Clarke, Jack C. Wileden, Leon Osterweil, and Alex L. Wolf. Foundations for the Arcadia environment architecture. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–13, Boston MA, November 1988.

[30] Wilhelm Schäfer, Burkhard Peuschel and Stefan Wolf. A knowledge-based software development environment supporting cooperative work. *International Journal on Software Engineering & Knowledge Engineering*, 2(1):79–106, March 1992.