An Architecture for Multi-User Software Development Environments

Israel Z. Ben-Shaul, Gail E. Kaiser and
George T. Heineman
Columbia University

ABSTRACT: We present an architecture for multi-user software development environments, covering both general and process-centered 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 that support collaborative work—with minimal effects on other components while still supporting integration. The architecture has been implemented for the MARVEL SDE, and we report our experience replacing and tailoring several parts of the synchronization component as part of MARVEL.

1. Introduction

Software Development Environments (SDEs) emerged in an attempt to address the complexity 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. Another key concern is how to support synchronization (or concurrency control) of the activities of multiple developers working on the same project. In traditional multi-user database applications, the classical atomic transaction model [11] synchronizes concurrent access to the database by isolating and serializing user activities, but there is wide agreement that classical transactions do not fit the requirements of collaborative work [56, 2, 44]. There have been numerous proposals for so-called cooperative transaction models (see [8] for a survey), but there is relatively little agreement on which particular approach to concurrency control is most appropriate for multi-user SDEs. We have therefore investigated flexibility and extensibility of concurrency control policies for multi-user SDEs from the system-architecture point of view, with the intention to ease replacement of the synchronization component for experimentation purposes and/or as greater consensus is achieved.

We consider both general and process-centered SDEs. Process-centered SDEs are a subclass of environments that provide a *process modeling language* in which an environment administrator defines the software development activities, their valid (partial) orderings, and any other constraints on their execution, collectively called the *process model*. The architectures of process-centered SDEs include process engines that "enact" the process model, a term used to encompass enforcement, automation and guidance of the users in carrying out the

process. Different organizations and/or projects may employ widely different processes using the same process-centered SDE. In contrast, a general SDE behaves the same way for every project. See [45, 40, 20].

The process modeling language and corresponding enaction engine must be extended with a notion of concurrency consistency and corresponding synchronization primitives (such as in [58]) to support multi-user process-centered environments, where the process model as well as the data is shared among multiple participants in the same process. In many process-centered SDEs, the process is defined in terms of rules and enaction is achieved through rule chaining. Examples include CLF [49], Oikos [1] and Merlin [62]. Such SDEs must support synchronization among automated chains of activities as well as activities directly invoked by users. In any multi-user SDE, 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 (henceforth 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 [35]. 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 [5, 7]. 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 sophisticated synchronization mechanisms that enact such models.

Section 2 gives the requirements that a MUSDE must fulfill, by definition, and some additional desired properties. Section 3 introduces the main characteristics and functionality of our architecture. Section 4 explains the rationale behind the architecture. Section 5 describes the implementation for MARVEL and our experience over the past two years, including changing and tailoring some parts of the synchronization component. Section 6 compares the architecture to related work. Section 7 briefly evaluates the architecture and summarizes the contributions of this research.

2. Requirements and Desired Properties

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 identification, ownership information, compilation status, etc. Product data may be integrated with control data (e.g., an object is defined as having state attributes representing "control" information 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 process, whereas in process-centered SDEs, control data reflects the state of the externally supplied process in progress.

Data consistency: A 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 coordinated together, since changes to product data must be propagated to dependent control data and vice versa.

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 perspective of the shared process. Again, this can be done in either a centralized or distributed fashion. For example, consider a constraint taken from the "ISPW problem" [29], 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 relevant participants in the concerted process.

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

Perhaps the most significant 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 cooperative transaction models, such as transaction groups [23], support this capability to a limited extent in that one hardwired policy is enforced among groups but different policies for within each group can be specified in a formalism supplied by the implementation [55]. 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 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 supporting *long-duration, interactive* operations and *cooperation* among concurrent operations. As noted in [10], any synchronization mechanism for a MUSDE must take into account that many activities in software development are long—the concurrency and failure atomicity of the conventional transaction model is not suitable, and interactive—response time is more significant than overall throughput. Cooperation implies sharing or exchange of partial information during collaborative development efforts.

Extensibility and broad scope of application: A MUSDE should be able to be extended with new tools, including commercial off-the-shelf tools not specifically developed for the MUSDE [19].

Visualization: A MUSDE implemented on a window-based platform should provide users with graphical visualization of both product and control data. 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 a MUSDE has to maintain up-to-date information as it is dynamically changed by multiple users.

Recovery: Recovery ensures consistency of persistent data in case of external and internal failures. Persistence of product data in principle can be provided by the host file system, but persistence of control

data must be provided by the MUSDE—which thus must also coordinate the dependencies between the two kinds of data. We distinguish between concurrency control, which is required by definition, and recovery, which is required in practice. These two functions are both carried out by the synchronization component.

All MUSDEs reflect to some degree the required properties outlined above, and a few support integration of arbitrary foreign tools, full visualization of both product and control data, and recovery beyond that supplied by the file system and individual tools. However, we know of no other environments that address the problem of *replacing* the synchronization component.

3. The Architecture

Two major principles underlie the overall design: *componentization* and *separation of mechanism and policy*. 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). We combine componentization with *layering*, which 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., Mach's replaceable pager [50]) and databases [61, 52], and layering has been followed in many areas such as communication protocols [17] and databases [13]. The combination seems especially promising for SDE technology, which is by nature subject to changes [59]. We suggest the potential to revise any system component (although with differing degrees of difficulty). Our major concern is to be able to modify the synchronization component with minimal effects on task management and data management. In particular, the synchronization subsystem should be separate from both components rather than tightly integrated into one or the other; such separation was previously promoted in Camelot for non-collaborative applications [22]. This kind

of component replaceability is termed horizontal adaptability, a subclass of structural static adaptability [12].

Our concern for separating mechanism and policy is deeply ingrained in the componentization. In particular, each component represents a class of mechanisms that can be tailored independently of other components. The project-specific policies are supplied in a range of different formalisms, depending on the component, from simple tables to sophisticated languages. This approach was inspired by the data modeling of database management systems and the process modeling of process-centered environments. We extended primarily to the synchronization component, to enable specification or parameterization of the cooperative transaction model on a per project basis.

The architecture is depicted in Figure 1, using terminology close to the "toaster" reference model [21]. We concentrate here on explaining *how* things work, and defer to section 4 the rationale for *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 on a local area network. The server and each client are implemented as distinct operating system processes. Each client represents a user *session* that lasts from invocation to exit, and acts as a front-end and an activity execution subsystem. The concept of *activity* encompasses all operations that manipulate product data, such as editing and testing, via internal or external tools; it does not include operations on control data (although these might occur in the server as side effects). Clients may spawn child operating systems processes to execute activities.

The server provides *data management, synchronization* 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 execute a specific activity. The server mediates access to both control and product data, and manipulates control data.

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

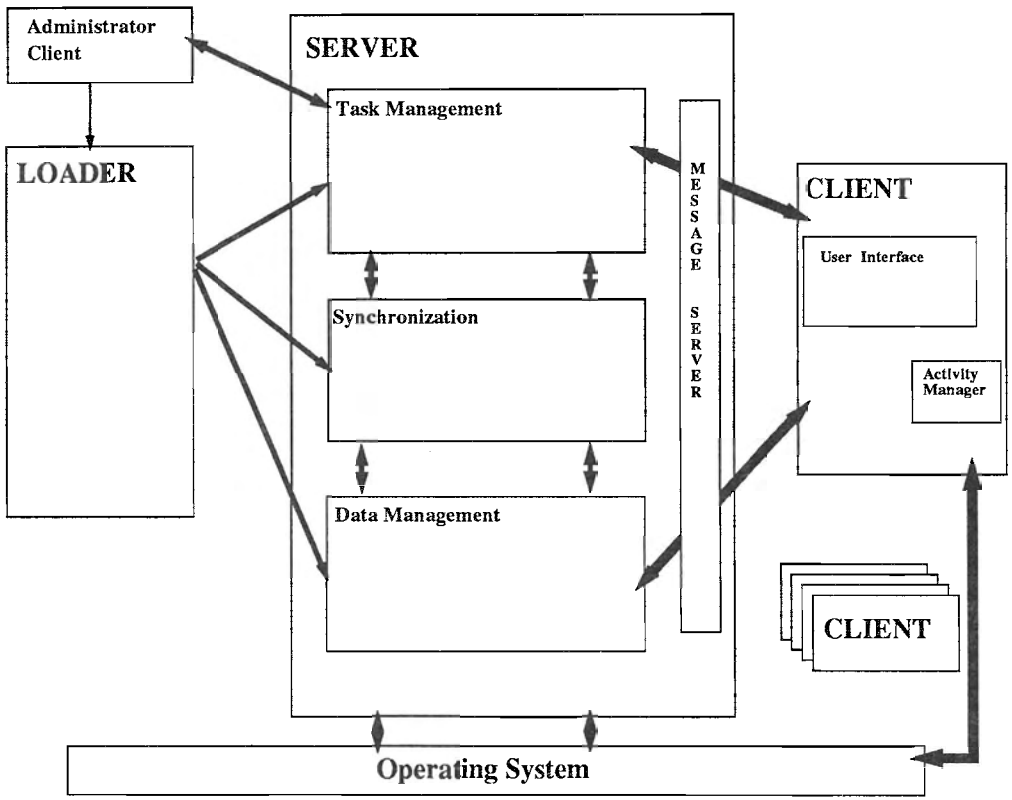

Figure 1: An Architecture for Multi-User SDEs

translator. Translation can be on-line, in which case the *Loader* acts purely as a parser for other interpretive components, or off-line, in which case it "compiles" the specifications into internal form.

The architecture distinguishes between normal users and an *environment administrator*. The administrator's role resembles that of a Database Administrator in conventional database management systems. 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 coordination model—the programmable aspects of the synchronization policy, if any.

3.1 Task Management

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 within the session (see below).

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 process segment to perform during that session [38].

Task Controller (TC): This is the central component of the SDE, 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 together 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 maintains a task context for each active task in the system.

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. Conflicts are detected by the Lock Manager, described below.

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 support flat transactions, or nested transactions that model the nesting of subtasks within a task [43].

TM-TC interface: The interface between TC and TM is a critical issue as it bridges between the task level and the synchronization level. It is desirable for TM to be independent of any specific task model and for TC to be independent of any specific synchronization mechanism, 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 concurrency control policies. However, semantics-based concurrency control inherently requires some knowledge of the task level to resolve conflicts that are context-sensitive [24]. This implies that the TC-TM interface may need to be augmented with a mediator component that reconciles information from both levels; MARVEL'S Coordination Manager (CM), which performs this role, is described in 5.6.

Lock Manager (LM): Conventionally considered part of the transaction manager, LM is treated in our architecture as a separate subcomponent. Its main role is to detect any potential violations of the data-consistency constraints, as defined by a lock-compatibility matrix. 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, so all such members of the group might simultaneously hold what would

normally be considered conflicting locks. ObServer [32] is a multi-user data server with a rich lock set, including communication modes (for notification), which is capable of supporting transaction groups. ObServer's communication modes can be implemented in LM with proper support from TM as part of conflict resolution; see section 5.6.

3.3 Data Management

Object Manager (OM): Implements the data model, provides persistence, and performs all requests for access and modification of both control and product data. For componentization to work, it is important that OM provide the upper layers with a data abstraction that avoids concern with internal representation. For example, upper layers should not know whether data is in main or secondary memory [54]. We assume a generic object-based data model with optional class ("is-a") hierarchy, composition ("is-part-of") hierarchy, and arbitrary relationships ("links") between objects. All, some or none of these might actually be supported by OM.

OM-LM interface: The main issue for the interface between OM and LM is whether data-consistency specifications need to be extended for a specific data model. For example, the existence of composite objects and semantic links among objects may mandate "intention" lock modes for ancestor and linked objects, respectively; the existence of types (classes) may similarly mandate intention locks as the object-based form of predicate lock [26]. Our architecture keeps the OM-LM interaction to the minimum: LM needs to know whether or not OM supports any such relationships, without concern for what the relationships are; OM must provide an abstract means for referring to the set of objects related to a given object, so that LM can place intention mode locks on those objects.

Storage and File Managers (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 files requested by other components (in a shared file system such as NFS only path names need be passed rather than the actual file contents). 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' file attributes and their file contents.

3.4 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 Manager (AM): Interacts with tools in an environment-specific manner. This might include spawning child operating system processes with suitable command lines and transforming data to/from objectbase and tool formats. There may or may not be communication between AM and the activity and between AM and the server while activity execution is in progress.

Client Command Processor (CCP): This subcomponent 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. CCP has no significant impact on the overall architecture.

Message Server (MS): Transfers information between the clients and the server over the communication medium. MS preserves 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. We divide task management between the clients and the server, where the clients are responsible for activity execution and the server for the rest. Since all long-duration operations are performed in the clients rather than

the server, task management does not become a bottleneck during parallel development.

3.5 An Example

Mika, a programmer, wants to change a source file to fix a bug. There are other active users working simultaneously on the same project. She logs into the SDE by invoking a client. The client initiates a connection with the server, passing along session variables. On the server side, SEM initiates a new session context. The client then receives a visual display of the objectbase for the user's screen, and Mika can then start to work.

She proceeds with a request **R** to edit a file **F**, represented by an object **O**. After preprocessing to check whether **O** exists and is unambiguous, the client sends the request to the server. When **R** is scheduled, SC updates SEM to point to the appropriate client's session and task contexts, and passes **R** to TC. TC tells TM to begin a transaction, which might be a subtransaction of some in-progress nested transaction when **R** is part of some on-going task. TC handles any task-related constraints for **R**, and then issues requests to TM to access the objects required, directly by the activity (tool) or indirectly by TC. TM requests the appropriate locks from LM for these objects. LM performs the lock requests by inspecting its previously placed locks and lock tables. If the request is initially denied due to a conflict detected by LM, TM resolves the conflict (e.g., by aborting or suspending some offending transactions, or by allowing apparently conflicting requests and insisting LM place the locks anyway, depending on the synchronization policy). If the locks are granted, LM then requests OM to access the objects. This may involve interaction with SM and FM. If all goes well, the relevant information propagates back up to this client, and SC selects the next client request.

AM then executes at the client in an environment-specific manner, for example, by spawning a child process to invoke a tool, pass input from the server, accept output and status code after the tool terminates, and—finally—report these back to the server. After SC restores the client's contexts, TC handles any task-related triggers for **R** (i.e., there may be task-related operations both before and after execution of each activity). This mode of interaction continues until there are no more activities to execute or until an exception occurs (e.g., a task is

aborted due to intervention of concurrency control or a failure), and the task is terminated. Notice that while a client is executing an activity, the server is not bound in any way to that client and can serve other clients. Thus response time is good, even if many of Mika's colleagues are simultaneously active.

4. *Alternatives, Decisions and Justifications*

4.1 Client-Server Separation

The first issue to consider is the degree of distribution of the MUSDE. 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-duration 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 and response time low. 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. These could be managed by the server as part of the session context, although the SDE would have to enforce compatibility among the definitions of process consistency [9]. Further, the server-client separation does not prevent distribution of the server 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 a relatively elaborate algorithm that decides how to resolve a conflict when it arises. This separation makes it possible 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 implementation of 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 data management. This separation contributes perhaps more than anything else to the flexibility of the system with respect to concurrency control. Examples that demonstrate this flexibility are given in section 5.

4.3 Tunable Lock Management

The alternatives are: (1) a non-locking policy, where concurrency control is optimistic; (2) a hard-coded lock set and lock-compatibility matrix; and (3) a programmable lock set and lock-compatibility matrix. Considering only options 2 and 3, the separation of LM from TM makes it impossible to predict what lock set and compatibility might be needed, and thus any hard-coded set would prove insufficient. 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.

Option 1 would involve replacing TM with an optimistic validation scheme [42], perhaps with a "merging" option as in NSE [31], and LM with a corresponding versioning mechanism. This should not affect the rest of the architecture, although obviously both TM and LM must be optimistic or both must be pessimistic. We have no direct experience in MUSDEs to back up this conjecture, but one of the authors has investigated the replaceability of optimistic concurrency control *vis a vis* locking in the context of concurrent programming languages

[48], and found that the same protocol could be employed with respect to the rest of the system.

4.4 Objectbase Visualization

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 either case control data is manipulated in the server, so the issue is not where to modify the data, but rather how to display it. Therefore, we can treat anything displayed at the client as a read-only replica. However, keeping entire replicas is unnecessary and may become prohibitively expensive as the number of clients increases. On the other hand, displaying only objects in current use by that client does not provide sufficient context to fulfill the "visualization" property.

We chose a compromise approach, where clients maintain a cache of the *structure* of the objectbase, mainly for reference purposes in selecting arguments to commands, but not the actual *contents* of objects. For each object, we maintain only 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.

5. Implementation for Marvel

The MARVEL 3.x architecture is illustrated in Figure 2. (We use the term 3.x to refer generally to all versions of MARVEL starting with 3.0; earlier versions of MARVEL supported only a single user.) 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 Loader is augmented by the Evolver, a data and process model evolution tool [36], which is outside the scope of this paper. The specifications that parameterize

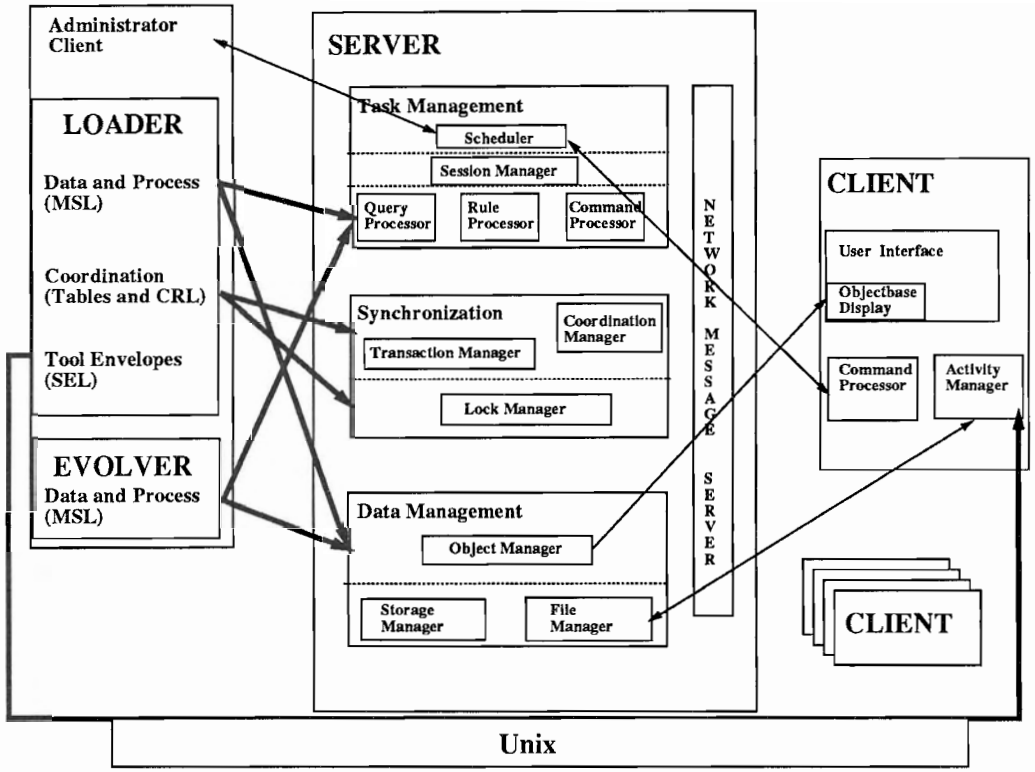

Figure 2: MARVEL 3.x Architecture

the various components—tool "envelopes" (see below) and data, process and coordination models—are written in various notations by the administrator and loaded 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 [37]. Each rule consists of a name; a list of typed parameters; a condition consisting of bindings to local variables and a complex property clause that must hold on the actual parameters and bound variables for the rule to fire; an optional activity that specifies a tool envelope and its arguments; and a set of mutually exclusive effects, each consisting of assertions to the objectbase that reflect one of 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* [30]. 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. If the condition is satisfied, the server sends sufficient information to the client for AM to execute the activity. When the activity is completed, RP asserts the effect indicated by the status code returned from AM 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 [6]. Consistency chains propagate changes and are by definition mandatory and atomic. Automation chains automate sequences of activities and are by definition optional; they may be terminated after any individual activity 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 a task, specifically each activity is transmitted to the client to carry out, at which point the server performs a context switch and turns to the next client request. That request might resume an in-progress task, following the completion of an activity in that client, 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*. 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 data structure is used for depth-first forward chaining.

One problem is that multiple instances of the same rule, with the same or different parameters, may be fired concurrently by the same or different clients. Since they all fire in the context of one RP (i.e., one address space), rules must be *reentrant*. Each invocation entails creation of a rule-frame, which consists of a pointer to the (read-only) rule definition and a dynamically allocated data section, which is retained throughout the entire life cycle of a rule chain.

5.3 Synchronization

TM supports a nested transaction model in which a task is modeled as a series of top-level transactions, each consistency chain is a subtransaction of the triggering transaction consisting of a further level of

subtransactions corresponding to individual rules, and each rule in an automation chain starts an independent top-level transaction. 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. Starting with MARVEL version 3.1, CM serves as a mediator between data and task management to permit relaxation of consistency under specified conditions. CM is discussed in section 5.6.

MARVEL'S composition hierarchy is based on ORION, using intention locks for ancestors [41]. 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 (e.g., deleting the 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 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, e.g., exclusive **X** takes precedence over shared **S**.

5.4 Data Management

One major consideration is the display-refresh policy for the objectbase image, from the viewpoint of OM. 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 TC message sent to each client. The third alternative was preferred as it saves communication overhead while keeping information reasonably up to date. An explicit "refresh" command is also provided for the user who wants to ensure his/her

image is up-to-date. We are working on a *delta* scheme, whereby batches of change records are transmitted rather than an entire image, when the time period or number of changes since the last refresh is smaller than a user-specified threshold.

SM uses the UNIX dbm package. Although more sophisticated data management strategies can be supported by dbm, SM loads the entire objectbase 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. Only file pathnames are stored in the objectbase, and the corresponding file contents are stored in "hidden" file system rooted at a distinguished directory specific to the objectbase. MS uses Internet sockets.

5.5 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. CCP includes the front end for an ad hoc query parser. AM is the most complex component of the client. It is in charge of spawning child processes for executing *envelopes*, basically shell scripts, for invoking the tools employed in the environment. AM 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 [27]. File attributes are supplied to/from envelopes as pathnames, so any content retrieval operates through the normal operating system interface entirely divorced from MARVEL.

The Loader generates a static rule network from the process model, which is used at runtime to determine chaining [34]. This network is loaded into RP, to define process consistency and opportunities for automation. 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 are loaded into LM, to specify potential conflicts, and a set of coordination rules are loaded into CM, to define semantics-based conflict resolution.

5.6 Experience

In this section we discuss our experience with the architecture as implemented for MARVEL, focusing on *changes* to the synchronization component since its initial implementation in Fall 1990. We also mention a few other changes that may be of interest.

In one specific MARVEL environment (see below), we started the compatibility-matrix with the standard shared and exclusive locks and corresponding intention locks (Figure 3), but then modified the lock tables several times. The current system includes five new lock modes, and changed compatibility for some old modes. One 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). So Strong Exclusive (**SX**) and Strong Shared (**SS**) locks were added, and **X** and **S** became compatible with any intention lock (Figure 4). The ancestor table was also 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 have been needed for MARVEL 3.1, where the specific lock modes to use for particular arguments of activities and rules are specified in another external table. Thus, a dramatic change in conflict detection can be achieved with very little overhead.

	S	X	IS	IX
S	y	n	y	n
X		n	n	n
IS			y	y
IX				y

S = Shared X = Exclusive
IS = Intention Shared IX = Intention Exclusive

Figure 3: Initial Compatibility Matrix

Whereas we implemented LM once and replaced its external tables several times, the entirety of the initial TM subcomponent was later replaced. The first TM implemented 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. The second instance of TM supported nested transactions. Each rule triggered during an automation chain, together with any consistency subchains emanating from it, is a top-level transaction that can be aborted without affecting other top-level transactions in the same task. This major change had no impact whatsoever on LM, and required only trivial changes to RP.

The final change to TM in MARVEL 3.1 added a hook for the Coordination Manager (CM). When TM would abort a transaction, it

	S	X	IS	IX	SS	SX	NT	NL	INT
S	y	n	y	y	y	n	n	y	y
X		n	y	y	n	n	n	y	y
IS			y	y	y	n	n	y	y
IX				y	n	n	n	y	y
SS					y	n	n	y	y
SX						n	n	y	n
NT							n	y	y
NL								y	y
INT									y

S = Shared, X = Exclusive
IS = Intention Shared, IX = Intention Exclusive
SS = Strong Shared, SX = Strong Exclusive
NT = Notification, NL = Nil
INT = Intention Notification

Figure 4: Revised Compatibility Matrix

requests assistance from CM. The administrator defines an optional set of *coordination rules* to specify scenarios when TM's default policy above may be relaxed, and prescribe appropriate actions in each such case. CM accesses an abstraction of RP's rule stacks, to try to match one of its coordination rules with the details of the current conflict scenario; if no match is found, then the decision reverts to TM. CM currently supports `notify`, `abort`, `allow` and `suspend` actions, and we are investigating the construction of a transaction algebra based on `split` and `join` operations [39].

The introduction of CM was coupled with one change to LM's tables, to add Notification (NT), Intention Notification (INT) and Nil (NL) locks. A notification lock conflicts with any other lock, so that CM is invoked whenever a notification lock has been or is being placed

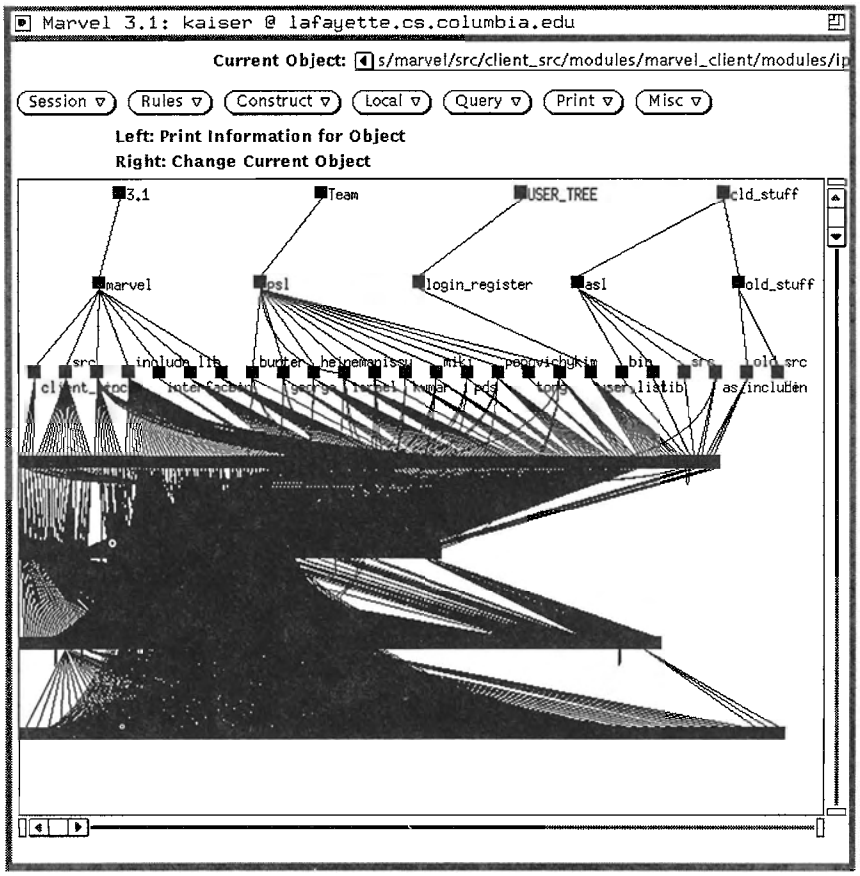

Figure 5: Top-Level MARVEL Objectbase Display

and another transaction requests or holds, respectively, a lock on the same object. The `notify` action informs the user who requested the notify lock, and then the conflict is `ignored`, meaning both of the "conflicting" locks are placed simultaneously and both transactions may proceed as if there were no conflict. The `ignore` action can also be used to support transaction groups, by permitting multiple transactions in the same group to proceed concurrently, even when there are serious conflicts (e.g., a strong shared lock and a strong exclusive lock).

The new XView client for MARVEL 3.1 allows manual switching among multiple in-progress tasks within a single client, while the earlier Xlib client permitted only one task at a time. Much of the purpose is to support the actions of the new CM, e.g., to suspend or abort

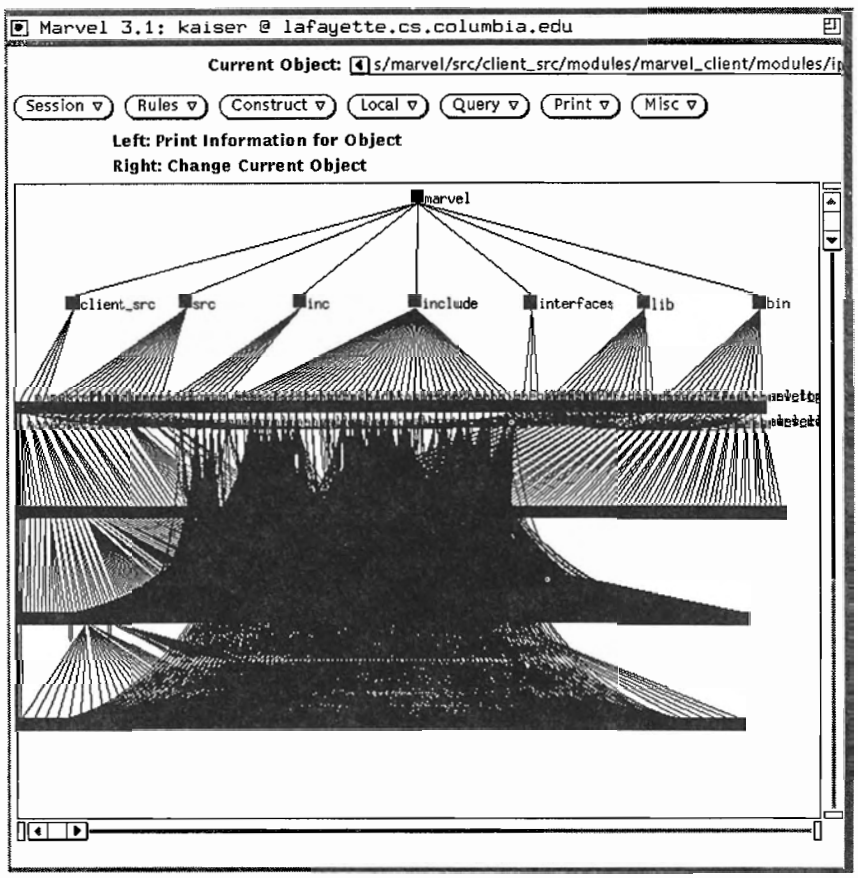

Figure 6: After Selecting Component AS New Display Root

in-progress activities; when such actions are applied to an Xlib client, they do not take effect until the user signals completion, perhaps wasting much human effort. Adding a choice among graphical clients required substantial changes to AM, and some to CCP, but only very minor changes to the server, mainly in the MS component.

AM has been enhanced to add "gray-box" integration for extensible tools. This interface allows repeated input and output while an activity is in progress. For example, a multi-buffer editor (such as emacs) makes it feasible for the user to request multiple files for editing incrementally, but our initial "black-box" approach required all inputs/outputs to be requested up front. Again, while major new functionality

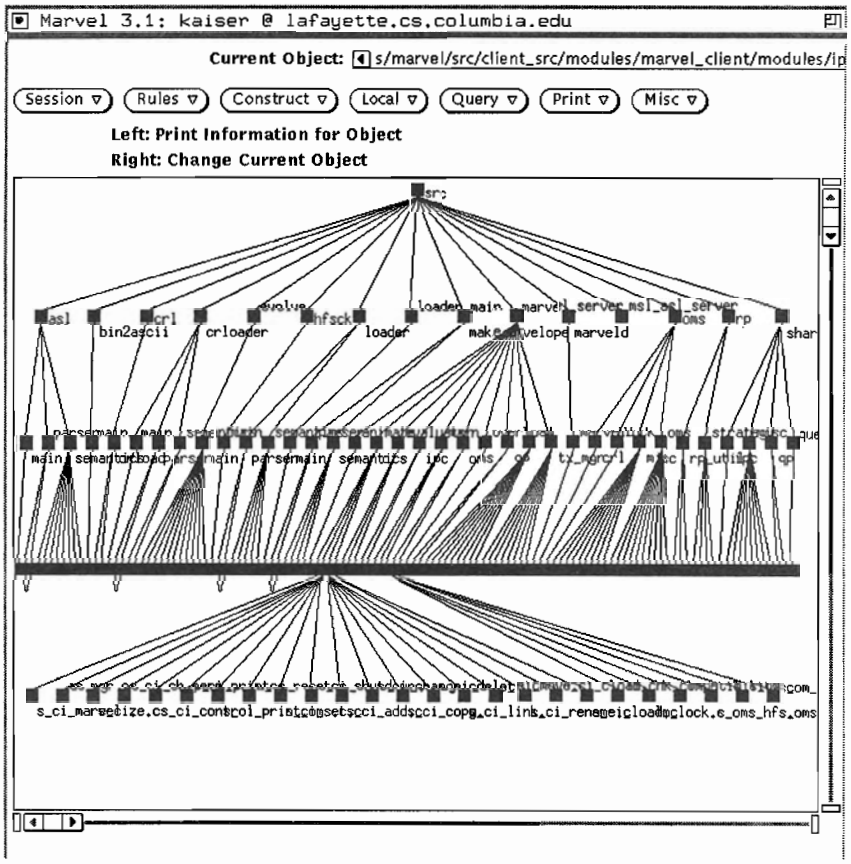

Figure 7: After Again Selecting Component AS New Display Root

was added to the client, very minor changes were required in the server.

The most extensive change to the client was performed as part of an experiment to treat MARVEL as a "policy tool" [25] for the Field broadcast message server [51]. The entire user interface was stripped off the client, CCP was modified to accept Field's string messages and convert them to the format required by the server, and AM was modified to transmit and register such messages [46]. This required only 950 new lines of code the MARVEL client and no changes at all to the server; 150 lines of Field's code were affected.

There have been numerous changes to TC, whose details are outside the scope of this paper. For example, in one experiment, we

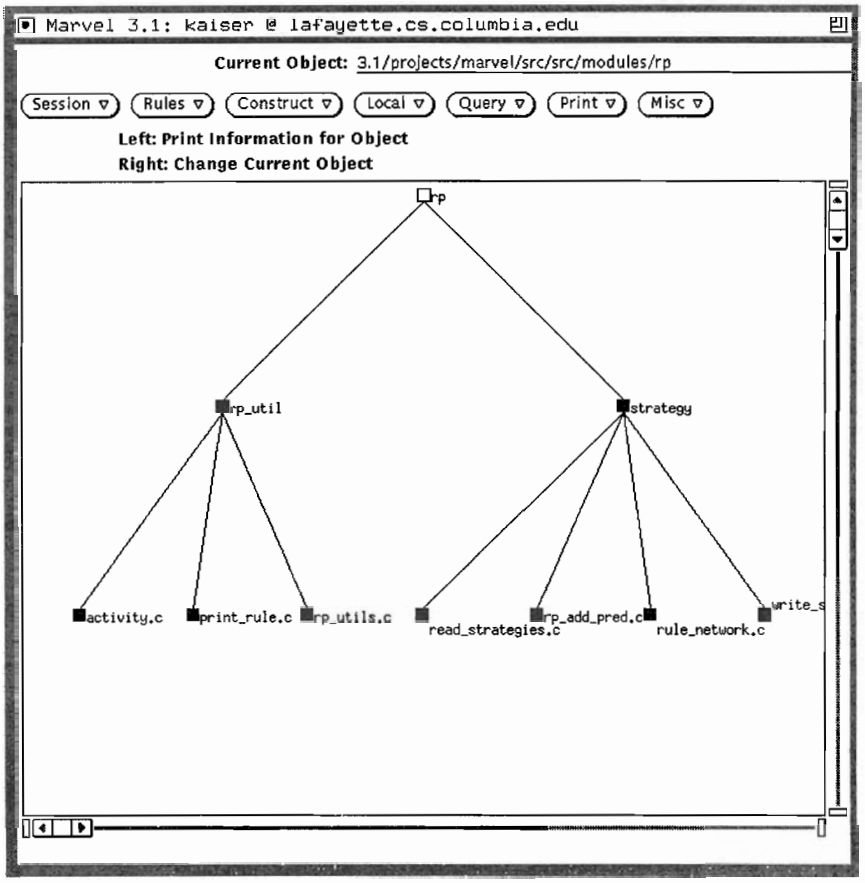

Figure 8: One More Click

developed a new process modeling language based on constrained expressions [3], and implemented it via an external translator into MARVEL's rules [38]. The combined language uses the two different formalisms to express global activity ordering and local constraints, respectively. No changes were required in the client, and the only change to the server was a 750-line module added to RP. Even though the new process modeling language explicitly expresses synchronization among multiple process participants, there were no effects at all on the transaction subsystem.

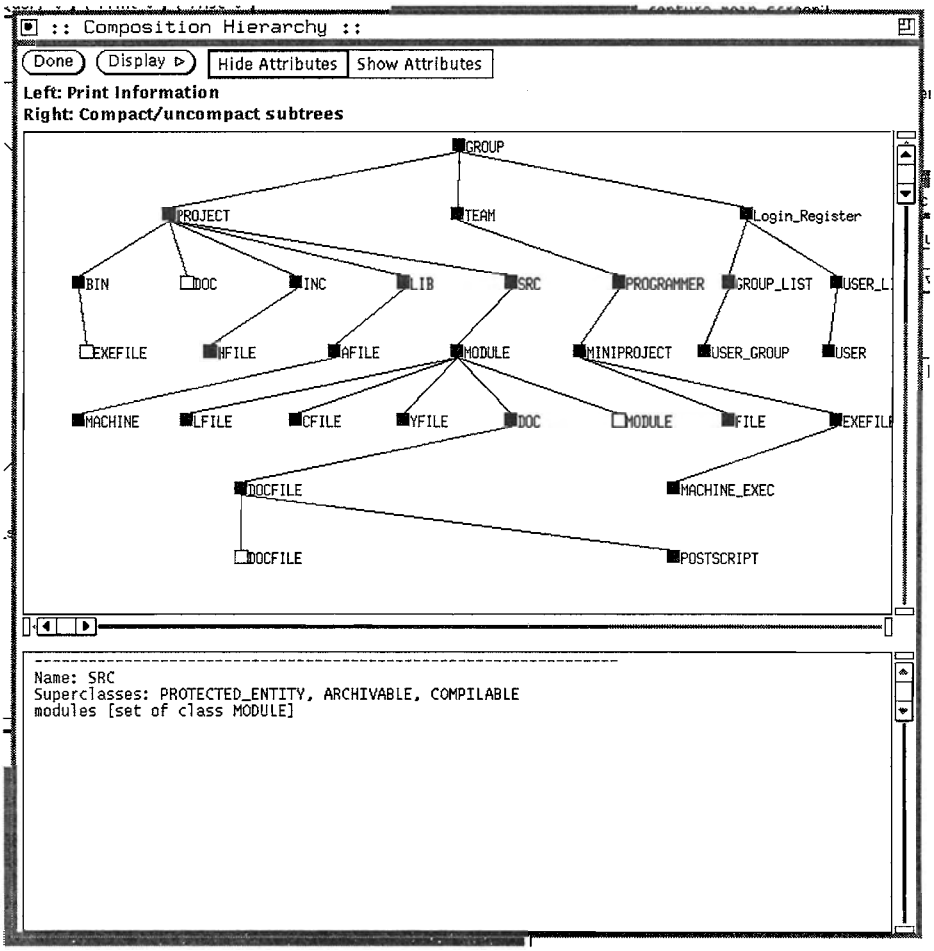

Figure 9: Separate Display of Objectbase Composition Definition

5.7 Status

MARVEL 3.1 is implemented in over 150,000 lines of C, lex and yacc. It runs on SparcStations (SunOS 4.1.2), DecStations (Ultrix 4.3), and IBM RS6000s (AIX 3.2), using X11R5 Windows, and comes with several hundred pages of administrator and user manuals. Earlier releases 3.0 and 3.0.1 have been licensed to about 25 educational institutions and industrial sponsors since December 1991. 3.0.1 was the first version fully developed and maintained using C/MARVEL, a MARVEL process for team programming in C, and C/MARVEL itself is now maintained using P/MARVEL, a MARVEL process for process model development and evolution [36].

6. Related Work

We mention only MUSDEs in this paper. See [47] for a survey of object management systems with particular concern for their synchronization facilities.

SMS (aka Gypsy) [16, 53] is an extended version control system that is tightly integrated with an object-oriented operating system. Synchronization is manual, and users work in isolation, each in his/her own "workspace". Although SMS 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.

NSE supports parallel and distributed development through an optimistic concurrency control model, which allows multiple users to access objects concurrently without locking them. When multiple users check in modifications to the same file, a tool aids in merging the multiple versions. File management is based on the concept of *environment*, a separate workspace with its own logical copy of the file system. Environments can be arbitrarily nested. This hard-coded synchronization policy dictates a methodology for software development, in which developers work primarily in isolation and conflicts are rare.

Arcadia [60, 33] is a process programming environment based on research in SDE technology underway by the Arcadia consortium.

Like our architecture, it is constructed from layered components that are intended to be replaceable. However, although process consistency from the process programming point of view is addressed extensively by Sutton [57], an independent synchronization component is conspicuously absent from the architecture. We guess that multi-user synchronization is provided by the object management system.

Melmac [18] is a process-centered environment that distinguishes between user-level and internal representations of the process. Unlike our architecture, Melmac's server is primarily concerned with data management and provides a simple concurrency control mechanism, and the clients are responsible for process enaction. One shortcoming evidenced by the examples given by Gruhn [28] is detaching process management from the server leads to an inability for rule chains to be interleaved even during activity execution—which might degrade response time significantly — so synchronization is not a major issue.

Oikos is a rule-based MUSDE that supports concurrency using a hierarchy of blackboards that resemble Linda's tuple spaces [14]. Oikos supports specification of a wide range of services available during 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.

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 [4]. Changes are grouped into evolution steps, which can be undone or redone [15].

Merlin is the closest system to MARVEL. From the process enaction 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 transaction model, using an object's state as determined by the rules in lieu of 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 chain-

ing operates in each user's working context (client), similarly to Melmac, 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 have had 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 such as our coordination manager—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, we envision the desire for explicit user interactions in variants of cooperative synchronization, e.g., to explicitly negotiate conflicting accesses. A special case of such an enhancement was developed for notification in MARVEL 3.1, but it is still not possible to define commands as part of the coordination model. To do so, we anticipate changes would be required in the two command processors 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. Some kind of server distribution seems warranted. 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 visualization without the overhead of maintaining complete objectbase replicas 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 synchronization 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 makes it possible 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 management and closer to the semantic, task level. We do not know of any other collaborative environment with such functionalities.

Acknowledgments

Naser Barghouti made numerous contributions to the MARVEL project in general and to its synchronization component in particular; Bill Riddle, Brian Nejmeh and Steve Gaede helped shape the functionality of multi-user MARVEL; Mark Gisi, John Hinsdale, Tim Jones, Raj Kumar, Yong Su Kim, Kevin Lam, Will Marrero, Hideyuki Miki, Steve Popovich, Moshe Shapiro, Peter Skopp and Mike Sokolsky participated in aspects of the implementation effort; Edmond Lee, Zhongwei Tong and students in the E6123 Programming Environments and Software Tools course helped by testing MARVEL as users. MARVEL 3.1 is available for licensing to educational institutions and industrial sponsors; send electronic mail to MarvelUS@cs.columbia.edu for information. An extended abstract of this paper was published in the *Fifth ACM SIGSOFT Symposium on Software Development Environments,* Tyson's Corner VA, December 1992, pp. 149-158.

Ben-Shaul is supported in part by NSF grant CCR-9106368. Kaiser is supported by National Science Foundation grants CCR-9106368 and CCR-8858029, by grants from AT&T, BNR, Bull, DEC, IBM, Paramax and SRA, and by the New York State Center for Advanced Technology in Computers and Information Systems. Heineman is supported in part by IBM and in part by the NSF Engineering Research Center for Telecommunications Research.

References

[1] V. Ambriola, P. Ciancarini, and C. Montangero. Software process enactment in Oikos. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments,* pages 183–192. Irvine CA, December 1990. Special issue of *Software Engineering Notes,* 15(6), December 1990.

[2] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *1st International Conference on Deductive and Object-Oriented Databases,* pages 40–57, Kyoto, Japan, December 1989. Elsevier Science.

[3] George S. Avrunin, Laura K. Dillon, Jack C. Wileden, and William E. Riddle. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions on Software Engineering,* SE-12(2):278–292, February 1986.

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

[5] Naser S. Barghouti, *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University, February 1992. CUCS-001-92.

[6] Naser S. Barghouti, Supporting cooperation in the MARVEL process-centered SDE. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments,* pages 21–31. Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes,* 17(5), December 1992.

[7] Naser S. Barghouti and Gail E. Kaiser. Modeling concurrency in rule-based development environments. *IEEE Expert,* 5(6):15–27, December 1990.

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

[9] Naser S. Barghouti and Gail E. Kaiser. Scaling up rule-based development environments. *International Journal on Software Engineering & Knowledge Engineering*. 2(1):59–78, March 1992.

[10] Philip A. Bernstein. Database system support for software engineering. In *9th International Conference on Software Engineering,* pages 166–178, Monterey CA, March 1987. IEEE Computer Society.

[11] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading MA, 1987.

[12] Bharat Bhargava and John Riedl. A model for adaptable systems for transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):433–449, December 1989.

[13] Alfred Brown and John Rosenberg. Persistent object stores: An implementation technique. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases Principles and Practice: The 4th International Workshop on Persistent Object Systems*, pages 199–212, Martha's Vineyard MA, September 1990. Morgan Kaufmann.

[14] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[15] Don Cohen and K. Narayanaswamy. A logical framework for cooperative software development. In Takuya Katayama, editor, *6th International Software Process Workshop: Support for the Software Process*, pages 69–71, Hakodate, Japan, October 1990. IEEE Computer Society.

[16] Ellis S. Cohen, Dilip A. Soni, Raimund Gluecker, William M. Hasling, Robert W. Schwanke, and Michael E. Wagner. Version management in Gypsy. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 201–215, Boston MA, November 1988. ACM Press. Special issues of *Software Engineering Notes*, 13(5), November 1988 and *SIGPLAN Notices*, 24(2), February 1989.

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

[18] Wolfgang Deiters and Volker Gruhn. Managing software processes in the environment MELMAC. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 193–205, Irvine CA, December 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[19] Mark Dowson. ISTAR—an integrated project support environment. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 27–33, Palo Alto CA, December 1986. Special issue of *SIGPLAN Notices*, 22(1), January 1987.

[20] Mark Dowson, editor. *1st International Conference on the Software Process: Manufacturing Complex Systems*, Redondo Beach CA, October 1991. IEEE Computer Society.

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

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

[23] 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.

[24] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems,* 8(2):186–213, June 1983.

[25] David Garlan and Ehsan Ilias. Low-cost, adaptable tool integration policies for integrated environments. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments,* pages 1–10, Irvine CA, December 1990. Special issue of *Software Engineering Notes,* 15(6), December 1990.

[26] Jorge F. Garza and Won Kim. Transaction management in an object-oriented database system. In *SIGMOD International Conference on Data Management,* pages 37–45, Chicago IL, June 1988. Special issue of *SIGMOD Record,* 17(3), September 1988.

[27] Mark A. Gisi and Gail E. Kaiser. Extending a tool integration language. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems,* pages 218–227, Redondo Beach CA, October 1991. IEEE Computer Society.

[28] Volker Gruhn. *Validation and Verification of Software Process Models*. PhD thesis, Forschungsberichte des Fachbereichs Informatik der Universität Dortmund, 1991. Bericht Nr. 394/91.

[29] Dennis Heimbigner and Marc Kellner. Software process example for ISPW-7, August 1991. /pub/cs/techreports/ISPW7/ispw7.ex.ps.Z available by anonymous ftp from ftp.cs.colorado.edu.

[30] George T. Heineman, Gail E. Kaiser, Naser S. Barghouti, and Israel Z. Ben-Shaul. Rule chaining in MARVEL: Dynamic binding of parameters. *IEEE Expert,* 7(6):26–32, December 1992.

[31] 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.

[32] Mark F. Hornick and Stanley B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Automation Systems,* 5(1):70–95, January 1987.

[33] R. Kadia. Issues encountered in building a flexible software development environment. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments,* pages 169–180, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes,* 17(5), December 1992.

[34] Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler, and Robert W. Schwanke. Database support for knowledge-based engineering environments. *IEEE Expert,* 3(2):18–32, Summer 1988.

[35] Gail E. Kaiser, Naser S. Barghouti, and Michael H. Sokolsky. Experience with process modeling in the MARVEL software development environment kernel. In Bruce Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences,* volume II, pages 131–140, Kona HI, January 1990.

[36] Gail E. Kaiser, Israel Z. Ben-Shaul, George T. Heineman, John K. Hinsdale, and Wilfredo Marrero. Process evolution for the MARVEL environment. Technical Report CUCS-047-92, Columbia University Department of Computer Science, November 1992. Submitted for publication.

[37] 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.

[38] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul. A bi-level language for software process modeling. In *15th International Conference on Software Engineering,* Baltimore MD, May 1993. IEEE Computer Society. In press. Available as Columbia University Department of Computer Science, CUCS-016-91, September 1992.

[39] Gail E. Kaiser and Calton Pu. Dynamic restructuring of transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications,* pages 265–295, San Mateo CA, 1992. Morgan Kaufmann.

[40] Takuya Katayama, editor. *6th International Software Process Workshop: Support for the Software Process.* Hakodate, Japan, October 1990. IEEE Computer Society.

[41] 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.

[42] H. T. Kung and John Robinson. On optimistic methods for concurrency

control. *ACM Transactions on Database Systems*. 6(2):213–226, June 1981.

[43] J. Eliot B. Moss. Nested transactions and reliable distributed computing. In *2nd Symposium on Reliability in Distributed Software and Database Systems,* pages 33–39, Pittsburgh PA, July 1982. IEEE Computer Society.

[44] Eric Neuhold and Michael Stonebraker (editors). Future directions in DBMS research. *SIGMOD Record,* 18(1):17–26, March 1989.

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

[46] Steven S. Popovich. Rule-based process servers for software development environments. In *1992 Centre for Advanced Studies Conference,* volume I, pages 477–497, Toronto ON, Canada, November 1992. IBM Canada Ltd. Laboratory.

[47] Steven S. Popovich and Gail E. Kaiser. An architectural survey of object management systems. *International Journal of Intelligent & Cooperative Information Systems,* 1993. In press. Available as Columbia University Department of Computer Science, CUCS-026-91, October 1992.

[48] Steven S. Popovich, Shyhtsun F. Wu, and Gail E. Kaiser. An object-based approach to implementing distributed concurrency control. In *11th International Conference on Distributed Computing Systems,* pages 65–72, Arlington TX, May 1991.

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

[50] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 31–39, Palo Alto CA, October 1987. Special issue of *SIGPLAN Notices,* 22(10), October 1987.

[51] Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software,* 7(4):57–66, July 1990.

[52] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in EXODUS. In *ACM SIGMOD Annual Conference on the Management of Data,* pages 208–249, San Francisco CA, May 1987. Special issue of *SIGMOD Record,* 16(3), December 1987.

[53] R. W. Schwanke, E. S. Cohen, R. Gluecker, W. M. Hasling, D. A. Soni, and M. E. Wagner. Configuration management in BiiN SMS. In *11th International Conference on Software Engineering*, pages 383–393, Pittsburgh PA, May 1989. IEEE Computer Society.

[54] Eugene Shekita and Michael Zwilling. Cricket: A mapped, persistent object store. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases Principles and Practice: The 4th International Workshop on Persistent Object Systems*, pages 89–102, Martha's Vineyard MA, September 1990. Morgan Kaufmann.

[55] Andrea Helen Skarra. *A Model of Concurrency Control for Cooperating Transactions*. PhD thesis, Brown University, May 1991.

[56] Michael Stonebraker, Lawrence A. Rowe, Bruce Lindsay, James Gray, Michael Carey, Michael Brodie, Philip Bernstein, and David Beech. Third-generation database system manifesto. *SIGMOD Record*, 19(3):31–44, September 1990.

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

[58] Stanley M. Sutton, Jr. A flexible consistency model for persistent data in software-process programming languages. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors. *Implementing Persistent Object Bases Principles and Practice: The 4th International Workshop on Persistent Object Systems*, pages 297–310, Martha's Vineyard MA, September 1990. Morgan Kaufmann.

[59] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. Language constructs for managing change in process-centered environments. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 206–217, Irvine CA, December 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[60] Richard N. Taylor, Richard W. Selby, Michael Young, Frank C. Belz, Lori A. Clarke, Jack C. Wileden, Leon Osterweil, and Alex L. Wolf. Foundations for the Arcadia environment architecture. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–13, Boston MA, November 1988. Special issues of *Software Engineering Notes*, 13(5), November 1988 and *SIGPLAN Notices*, 24(2), February 1989.

[61] David L. Wells, Jose A. Blakely, and Craig W. Thompson. Architecture of an open object-oriented database management system. *Computer*, 25(10):74–82, October 1992.

[62] Burkhard Peuschel Wilhelm Schäfer 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.