Integrating an Object Server with Other Worlds

ALAN PURDY and BRUCE SCHUCHARDT
Servio Logic Development Corporation
and
DAVID MAIER
Servio Logic Development Corporation and Oregon Graduate Center

Object-oriented database servers are beginning to appear on the commercial market in response to a demand by application developers for increased modeling power in database systems. Before these new servers can enhance the productivity of application designers, systems designers must provide simple interfaces to them from both procedural and object-oriented languages. This paper first describes a successful interface between an object server and two procedural languages (C and Pascal). Because C and Pascal do not support the object-oriented paradigm application, designers using these languages must deal with database objects in less than natural ways. Fortunately, workstations supporting object-oriented languages have the potential for interacting with database objects in a much more integrated manner. To integrate these object-oriented workstations with an object server, we provide a design framework based on the notion of workstation *agent* objects representing *principal* objects in the database. We distinguish two types of agents: *proxies*, which forward most messages to the principal objects, and *deputies*, which can cache state for their principal and act with more autonomy. The interaction of cache, transaction, and message management strategies makes the implementation of deputies a nontrivial problem. The agent metaphor is being used currently to integrate an object server with a Smalltalk-80™ workstation.

Categories and Subject Descriptors: C.2.4 [**Computer-Communications Networks**]: Distributed Systems—*distributed applications*; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*; H.2.0 [**Database Management**]: General; H.2.1 [**Database Management**]: Logical Design—*data models*; H.2.4 [**Database Management**]: Systems

General Terms: Design, Languages

Additional Key Words and Phrases: GemStone, object-oriented environment, object server, Smalltalk-80

1. INTRODUCTION

During the last three years a team at Servio designed and implemented an object-oriented database server (or object server) called GemStone[1] [7–9]. GemStone delivers to application developers a database subsystem with a Smalltalk-like object model instead of one of the more traditional record-oriented models (e.g.,

[1] GemStone is a registered trademark of Servio Logic Development Corporation.

Authors' addresses: A. Purdy, Xerox PARC/Northwest, 10220 S.W. Greenburg Rd., Portland, OR 97223; D. Maier, Servio Logic Development Corp., 15025 S.W. Koll Parkway, Beaverton, OR 97006; B. Schuchardt, Servio Logic Development Corp., 15025 S.W. Koll Parkway, Beaverton OR 97006.

relational, hierarchical). This object model allows applications to manage information (e.g., documents, pictures, sound) not easily handled by more traditional database systems.

The GemStone server provides neither graphics nor terminal handling support for building end-user interfaces to database applications. Instead of trying to support fast-response user I/O from a central machine, we assume that application development and application use is centered at a personal workstation such as an IBM-PC[2] or a Smalltalk machine (Figure 1). The workstation, in turn, communicates with a GemStone object server (and other servers) over a network. The workstation's dedicated processing power provides the capabilities required for high-quality interfaces to application users (e.g., bit-map graphics, mouse, windows), while the object server provides modeling power and efficient sharing of persistent objects. The computing environments on these workstations represent the "other worlds" with which the GemStone object server must be integrated.

We begin this paper with a review of the GemStone system, covering the motivation for GemStone, the object model, the OPAL[3] language, and GemStone's database features. We then describe an interface to GemStone from the "procedural world" of a language such as C or Pascal. This interface supports both the application development tools delivered with GemStone and application programs running on the workstation.

The GemStone system architects ultimately wish to provide a single "seamless" environment to application developers working in an "object-oriented world," such as Smalltalk, and wishing to use GemStone's features. Although integrating GemStone with another "object world" would seem easier than integrating it with a "procedural world," this task is more complex than it first appears. We discuss the requirements for a seamless integration and then present a design framework based on passive and active "agents" [1, 15] for GemStone objects. We conclude with a discussion of our progress toward the "seamless" goal, and the open research issues concerning a seamless integration.

1.1 Motivation for GemStone

The limitations imposed by the record-oriented data model of most database systems severely restrict the kinds of information that can be easily managed by these systems. GemStone was designed to greatly increase the data-modeling power of a database component with the hope of vastly reducing the development time of applications with complex information needs. Such applications include office information systems, computer-aided design, documentation of complex mechanical systems (automobiles, airframes), and artificial intelligence knowledge bases. Such a system's data model should support the definition of new data types, rather than constrain designers to a fixed set of predefined types. Also, the bounds on the number and size of data objects should be determined only by the amount of secondary memory, not primary memory. Such a system should also provide shared access to persistent data in a multiuser environment with the usual database amenities such as concurrency control, recovery,

[2] IBM-PC is a registered trademark of IBM Corporation.

[3] OPAL is a registered trademark of Servio Logic Development Corporation.

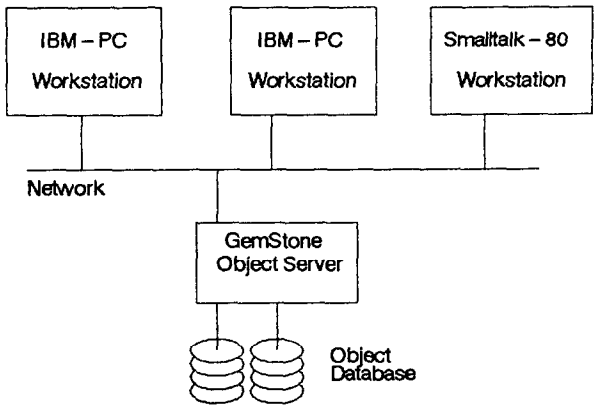

Fig. 1. System topology.

authorization, and system management functions. Finally, the system should have an interactive interface for defining new database objects, writing database routines, and executing ad hoc queries.

By combining the capabilities of an object-oriented language with the storage management functions of a traditional data management system we assumed that we would meet these goals and provide an environment that would reduce application development effort and promote data sharing among applications.

1.2 GemStone Object Model

GemStone's object model is identical to that of Smalltalk-80[4] [3]. The three principal concepts of the GemStone object model are *object*, *message*, and *class*. These correspond roughly to *record*, *procedure call*, and *record type* in conventional systems. An object is a private memory with a public interface. The private memory of (most) objects is structured as a list of *instance variables* containing their values. Instance variables are like field names in a record or indices in an array; their values are references to other objects. Objects communicate with other objects only by passing messages among themselves. These messages are requests for an object to change its state, to return a value (i.e., another object), or to perform some sequence of actions. The set of messages to which an object responds (the public interface) is called its *protocol*. An object may only inspect or change another object by sending a message to it. Each object responds to a received message by executing a *method* written in the OPAL language. Objects sharing the same format and methods are grouped into a *class* and called *instances* of the class. The methods and format of a class's instances are factored and stored once in a single object describing the class, the *class-defining object*. Each instance of a given class contains a reference to its class definition. The classes in GemStone are arranged in a *class hierarchy*, with each subclass inheriting behavior and structure from its superclasses. When a message is received by an object, the object consults with its class (and potentially its superclasses) to locate the proper method for execution.

[4] Smalltalk-80 is a registered trademark of Xerox Corporation.

1.3 OPAL Language

The OPAL language syntax and semantics are nearly identical to the Smalltalk language presented by Goldberg and Robson [3]. OPAL and Smalltalk share two major language constructs: message expressions and method definitions. Each is discussed below.

1.3.1 *Message Expressions.* All message expressions look like ⟨receiver⟩ ⟨message⟩. Each receiver is either a variable identifier, a literal, or another expression denoting an object that receives and interprets the message. Each message contains a *selector* (a procedure name) and possibly message *arguments*. When a message expression is executed, another object is returned as the value of the message expression. Thus a message receiver may be an expression that, when evaluated, returns an object that is then sent another message.

OPAL supports three kinds of messages: unary, binary, and keyword. Unary messages have no arguments and have a single identifier as a selector, for example,

```
5 negated
```

This expression sends the unary message composed of the unary selector `negated` to the object 5 and returns the object whose value is −5. Binary message expressions have a receiver, one argument, and a message selector that comprises one or two nonalphanumeric characters. For example, to multiply 8 by 3, we send the message "* 3" or "multiply by 3" to the receiver "8":

```
8 * 3
```

This binary expression returns the object 24. In this example, the binary selector * is used for multiplication. Comparisons are binary messages too:

```
emp1 salary <= emp2 salary
```

This binary expression demonstrates precedence. The two unary messages with the selector `salary` have precedence over the binary message whose selector is <=. Thus, the salaries of `emp1` and `emp2` are compared with the binary message. Keyword messages have one or more arguments and multipart selectors composed of alphanumeric characters and colons. For example, to set the third component of an array held in anArray to the string 'Ross', we use

```
anArray at 3 put: 'Ross'
```

In this example, the selector is read as "`at:put:`" with the two arguments of 3 and 'Ross'.

1.3.2 *Methods.* Methods define all execution in GemStone. Each method corresponds to one message selector and is defined within the scope of the class instance that is the receiver of the message. Thus, a method can directly access the named instance variables of the receiving object. A method (like a Pascal function) has a formal declaration, an optional declaration of temporary variables, and a body composed of a sequence of OPAL message expressions. The method body always returns a value to the sender of the original message. The following example defines a method whose unary selector is `wholeName`. This method first defines a temporary variable named `temp`. The method body assigns an instance variable named `first` to the temporary, then concatenates the

temporary, a blank, and the value of the instance variable `last`. Finally, it returns the result of that concatenation to the sender of the message.

```
wholeName
  | temp |
  temp := first.
  ↑ temp + ' ' + last
```

1.4 GemStone Database Features

GemStone combines the database management features common to most commercial database systems (e.g., Ingres [13], IMS [5]) with the object-oriented data model provided by the Smalltalk-80 system. GemStone's main features are

Sharing of Objects. GemStone provides each user with a distinct list of dictionaries called a **symbolList**. Although this list is private to each user, the dictionaries it contains can be shared. Thus a dictionary shared by two users allows the sharing of the objects contained in that dictionary. A dictionary is a collection of key-value pairs and supports the naming of objects. In this context a dictionary acts much like a file directory.

Resilience to Common Failure Modes. The data in a database often represent a significant capital expense. Most database users, including users of GemStone, expect their databases to survive all power failures that do not permanently damage the secondary storage media. In addition, if the reliability of the disk drives is insufficient, users can selectively replicate the stored objects on line, ensuring that the database survives single-point disk failures.

Multiple Concurrent Users. The standard mechanism for sharing a database requires the concept of a transaction. Intended changes to a database become visible to others only when a user successfully commits changes. GemStone uses an optimistic concurrency control policy [6].

Security. Database systems must secure information from unauthorized change or access. GemStone secures the object database by first authenticating each user through a user name and password. Along with user authentication, groups of objects may be explicitly marked as either read only, read/write, or no privileges for selected users.

Centralized Server. GemStone is a centralized server for a database of objects. It currently does not allow a database to be distributed among several GemStone servers.

Primary and Secondary Storage Management. GemStone hides from application designers the paging of objects between secondary and primary memory, and supports objects larger than what can fit in the server's primary memory.

Method Execution. GemStone supports a Smalltalk-like execution model on the server. This capability gives the application designer using GemStone a choice: Copy an object's state to the workstation for manipulation or execute messages remotely on the GemStone server.

Uniform Language. GemStone presents one language, OPAL [7, 8], to its users. Through OPAL the user manipulates the information in the database (Data Management Language), defines new classes (Data Definition Language), writes portions of application programs (General Computation Language), and controls the GemStone server (System Command Language).

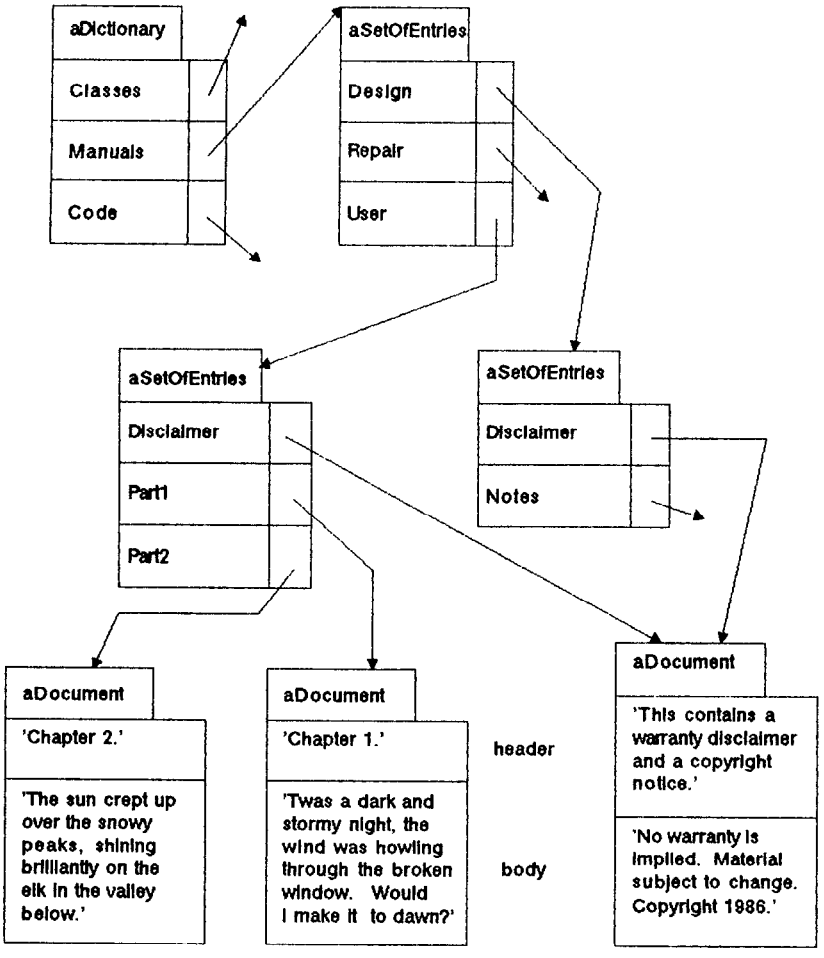

Fig. 2. Structured documents example.

Fast Associative Access. Database systems are traditionally efficient at finding all members of a set meeting a selection criteria. GemStone allows users to dynamically add (or remove) associative access structures to accelerate such membership tests [9].

1.5 An Example Application

To illuminate the ideas in this paper, we present a simple document archiving system. For this example a set of documents is stored on GemStone in a user-specific dictionary. The application accesses this set to find a given document, update a document, and add a new document. A document set is hierarchical: It can contain both documents and other sets of documents. A user can move documents between sets and can have the same document appear in multiple sets. The main classes in this example are `Dictionary`, `SetOfEntries`, and `Document`. A `Dictionary` is a subclass of `Set` and stores an unordered collection of `Associations` between a name and any arbitrary object.

`SetOfEntries` is a subclass of `Array` that stores a list of `Associations`. `SetOfEntries` differs from `Dictionary` in that it maintains a user-specified order on its entries. Each entry in a `SetOfEntries` associates the name of an entry with the value of an entry. In our example, the value will be either an instance of `Document` or another instance of `SetOfEntries`. A document has two instance variables, `header` and `body`; both are instances of `String`. A more complete system would impose further structure, such as headings, sections, and paragraphs, on the body of a document. Figure 2 illustrates a database containing pieces of manuals. Note that one of the documents is shared by two entries in `Manuals`.

2. INTEGRATING GEMSTONE WITH A PROCEDURAL ENVIRONMENT

To support applications in the "procedural world," GemStone provides a C and Pascal callable object module (Figure 3), which the designer links with applications running on an IBM-PC. This Procedural Interface Module (PIM) implements remote procedure calls [14] to functions supplied by the GemStone object server. The PIM hides the network communications protocols and provides calls to GemStone for controlling sessions, transporting object states between the two environments, and sending messages to objects residing on GemStone. The calls in each category are described in Tables I–III. Although these descriptions somewhat simplify the actual PIM functions supplied by GemStone, the essentials are preserved.

The session-controlling functions establish a connection between a workstation program and GemStone, control GemStone sessions, and control GemStone transactions. The other categories of functions allow workstation procedures to reference GemStone objects by unique identifiers that are represented in C or Pascal by variables of type *Oop* (for Object-Oriented Pointer). Before sending a message to a GemStone object, the workstation must first obtain the identifiers of the receiving object, the desired message selector (which will be an instance of the class `Symbol`), and the message's arguments. To use the object transporting functions to access information stored on GemStone, a C or Pascal application designer must know how to interpret GemStone's objects when their state is "fetched" into the application process on the PC. Knowledge of the following object formats is sufficient for building applications: Number, Character, Boolean, UndefinedObject, byte objects, pointer objects, and class-defining objects. All the functions take a session identifier as an argument, which we have omitted for simplicity.

For the primitive GemStone classes of SmallInteger, Character, Boolean, and UndefinedObject, the state (or value) of an instance is directly encoded in its Oop. The PIM includes utility functions to convert between C or Pascal values and their equivalents for instances of these classes. The PIM also includes functions to convert between GemStone floating-point and large integer values and their Pascal or C equivalents. Applications access information about other GemStone objects by sending messages to objects (for execution on GemStone) or by fetching object states from the object server into the Pascal or C address space. An object's state will be formatted either as an array of bytes (e.g., instances of `String` or `Symbol`) or as an array of Oops, each of which identifies

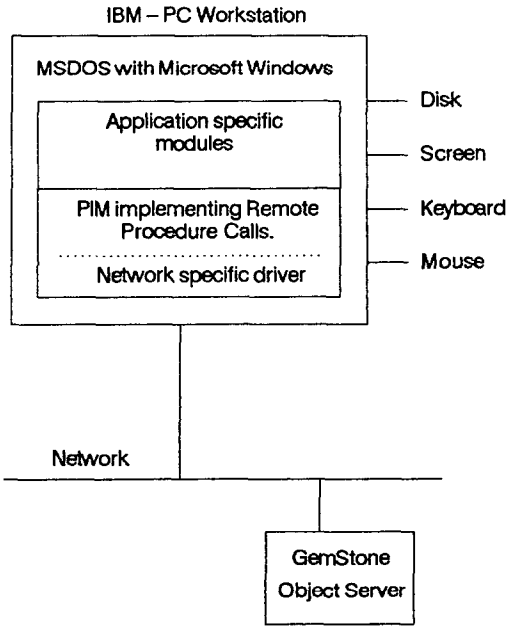

Fig. 3. Workstation details showing PIM.

Table I. Session Controlling Functions

`Logon(UserId, Password)`	Create a virtual circuit to the object server, and authenticate the user. The PIM hides from the application the details of the actual communications protocol
`Logoff`	Abort the current transaction, log the user off the server, and disconnect the session's virtual circuit
`AttemptCommit`	Attempt to commit the changes since this user's last transaction. If the transaction succeeds, true is returned; otherwise, false is returned
`Abort`	Tell GemStone to discard all the intended changes to the database since this transaction began. Begin a new transaction
`SignalInterrupt`	This function is an asynchronous action that signals the server to stop whatever it is doing for this user and await further instructions
`Resume`	Tells the suspended GemStone session to continue from where it was suspended

another GemStone object. Although giving an application direct access to the internal state of a GemStone object violates the integrity of an object, such access is necessary for efficient copying of objects between GemStone and the workstation. For example, without these transporting functions, access to each character in a `String` instance would require a separate PIM call. Because class definitions and methods are themselves objects and respond to the same functions as other objects, the PIM supports class and method definition. In practice, a designer rarely defines these directly through the PIM. Classes and methods are more

Table II. Remote Message Sending Functions

`SendMessage(ResultId, ReceiverId, MessageId)` Send the given message to the indicated receiver object. The server executes the indicated method and returns the result. Each argument to SendMessage is in the form of a GemStone object identifier. Internal to the IBM-PC, these are represented as 32-bit values
`ExecuteStatements(ResultId, StatementsId)` Send a sequence of OPAL statements in text form to the object server, which compiles it, executes it, and returns the result

Table III. Object Transporting Functions

`Fetch(ObjectId, From, To, Buffer)` Copy the given object's state from the server to the application process buffer. The caller may selectively move a fragment of the object
`FetchInfo(ObjectId, Size, ClassId, Form)` Return an object's metainformation to the application process, including its size in bytes or pointers, its class, and its format (bytes, pointers, atoms)
`Store(ObjectId, From, To, Buffer)` Copy the given object's state from the application process' buffer to the server. The caller may selectively move a fragment of the object
`Instantiate(ClassId, NewObjectId)` Create a new object of a given class and return its object identifier to the application

often defined through the browser in the OPAL Programming Environment or through some other application written on top of the PIM.

2.1 Example of PIM's Use

This section provides Pascal code fragments for the application described above in Section 1.5. This application accepts from the user a path to a document such as `User/Part1`, copies the given document's body to a local file called `MyManualBody` for editing, then moves the edited string back to GemStone as the document's new body before committing the transaction. The OPAL compiler accesses a user's `symbolList` (see Section 2.4) when compiling OPAL source code submitted by the user. We assume that all users of this application have an object named `Manuals` in one of their `symbolList` dictionaries. Thus any OPAL code sent to `ExecuteStatements` that mentions `Manuals` will obtain an object corresponding to that name.

```
{Include the PIM external declarations}
PROCEDURE TransferString(AnOop: OOP; Size: INTEGER; FileName:
STRING);
{Move a string from a GemStone database to a local file}
 CONST MaxChunk = 512;
 TYPE
  ChunkRange = 1 . . MaxChunk;
  Chunks = ARRAY[ChunkRange] OF BYTE;
 VAR
  Next: INTEGER; Chunk: Chunks;
  ChunkSize: ChunkRange; OutFile: FILE OF Chunks;
```

```
BEGIN
 Next := 1;
 ChunkSize := MaxChunk;
 OPEN(OutFile, FileName);
 WHILE Next <= Size DO BEGIN
  IF(Size-Next + 1) < MaxChunk THEN
   ChunkSize := Size-Next + 1;
  Fetch(AnOop, Next, Next + ChunkSize - 1, Chunk);
  Write(OutFile, Chunk: ChunkSize);
  Next := Next + ChunkSize;
  END;
 CLOSE(OutFile);
END;

PROCEDURE ExampleProgram;
 VAR Body, Class: OOP; Size, Form: INTEGER; Path: STRING;
     Success: BOOLEAN;
BEGIN
 IF Logon('Alan', 'SwordFish') THEN BEGIN  {Logon with Id, Password}
 ReadLn(Path);
 ExecuteStatements(Body, '(Manuals find. ''' + Path + ''' )  body');
```

{The preceding assumes the existence on GemStone of a method named **find:** for a **SetOfEntries** that takes a path argument (such as 'User/Part1') and returns the appropriate object. It also assumes that a **Document** responds to **body** by returning the body's string.}

```
  FetchInfo(Body, Size, Class, Form);
  IF Class = StringClassOop THEN BEGIN
   TransferString(Body, Size, 'MyManualBody');
   {... Edit the Body}
   {... Transfer it back to GemStone}
   END;
   Success := AttemptCommit;  {Try to commit the changes}
  Logoff;
  END
END;
```

2.2 A Critique of the PIM in Practice

Servio found the Procedural Interface Module adequate for building the applications that constitute the OPAL Programming Environment (OPE) that runs on the IBM-PC under Microsoft Windows.[5] The OPE includes

—*class browser*, which allows a user to examine, add, and modify GemStone class and method definitions (this browser is similar to the class browser in the Smalltalk-80 programming environment [4]);

—*bulk loader/dumper*, which allows a user to transfer formatted data with fixed record types between PC-based files and GemStone;

—*workspace editor*, which allows a user to enter, edit, and execute OPAL expressions.

[5] Windows is a registered trademark of Microsoft Corporation.

Each of these OPE applications accesses information stored on the GemStone object server. In each case the designer wrote the user interface code for the application in Microsoft C, accessed GemStone through the PIM, and accessed the user through Microsoft Windows. Once our designers became familiar with these tools, the delivery of a new application often became primarily an exercise in specification. For example, a document archive application similar to the preceding example took one developer three days to design, code, and test.

This ease of development is not without a high training and education cost. The GemStone object server requires the application designer to be familiar with GemStone's PIM, its OPAL language, transactions, and other concepts of database management. Although Microsoft Windows vastly improves the quality of an application's human interface, windows imposes its own long learning curve.

The option of two execution environments (PC or GemStone) complicates the application design process: The designer must decide whether to copy an object to the PC for processing or to forward messages for execution on the server. This decision is rarely simple. It often requires an intimate understanding of the relative speed of the two execution environments, the bandwidth connecting the two machines, and the degree of competition for that bandwidth. GemStone offers relatively fast access to the structure of objects, but it is slower than most implementations of Smalltalk for general computation. Thus, extensive arithmetic and string operations are best performed on the PC. The path parsing code we referenced above (`find:`) would probably execute more quickly on the PC if we ignored the time required to transfer instances of `SetOfEntries`. Because of this transfer time, however, the `find:` function would probably take less clock time if it executed on GemStone. Also, when an object's state is copied to the PC, the application designer must ensure that any changes to that copy are transferred to GemStone at the appropriate point (e.g., prior to committing a transaction).

The PIM does not provide certain functions needed by most applications. For example, the PIM lacks the following:

Control of Cached GemStone Objects. If the object states cached in the PC were managed directly by the PIM, then two modules of an application accessing the same object would access the same state. As the PIM currently stands, if two modules each cache object states, the designer must guard against the same object being cached twice. This is especially important if both modules are modifying objects in their cache.

Functions to Transfer Large Objects. Applications often move large objects between local files on the workstation and the GemStone object server. The example above demonstrates the kind of function needed.

Transitive Transfers. Because a GemStone object can be composed of references to other GemStone objects, we also need standard procedures for transporting all the objects transitively reachable from a given object.

Before we provide these functions in the "procedural world," we first wanted to prototype them in a Smalltalk-80 integration with GemStone. Smalltalk's

"object world" is more supportive of such experimentation. In the next section we discuss the design and prototype implementation of this integration.

3. INTEGRATING THE GEMSTONE AND SMALLTALK-80 ENVIRONMENTS

The task of integrating GemStone with the Smalltalk-80 environment appears easy because both share similar data models and similar languages. However, this task offers several difficult challenges. In this section of the paper we first define the goals of such an integration, then describe a design framework based on *agents*, which are Smalltalk representatives of GemStone objects. These agents are of two kinds: *proxies*, which are little more than a transparent packaging of the PIM remote message sending functions for the Smalltalk environment, and *deputies*, which take a more active role in implementing various policies for caching the state of GemStone objects in Smalltalk's object memory. Pasco's encapsulators [10] have characteristics similar to this agent model.

3.1 Integration Goals

We want to provide developers with the illusion of one uniform application development environment quite unlike what the PIM provides to Pascal and C developers. Such a "seamless" integration would combine Smalltalk's development environment with GemStone's ability to easily access and modify persistent, shared, and secure objects. This integration has the following specific goals:

Object Transparency. Object transparency allows developers to design and implement applications without caring where an object is located or where a method is executed. Object transparency frees most developers from dealing with difficult caching and communication issues that arise when transporting objects or messages between Smalltalk's object memory and GemStone's stable storage. For example, the method that adds a new entry to a `SetOfEntries` in the document archive example should not need to test whether the `SetOfEntries` is cached in the Smalltalk workstation.

Automatic Database Object Creation. Objects created in the Smalltalk environment should continue as Smalltalk objects until they are referenced by a GemStone object. All Smalltalk objects referenced by a GemStone object should be converted automatically to GemStone database objects. In our example application a seamless system should automatically convert a new Smalltalk document to a GemStone document when it is added to a GemStone `SetOfEntries`.

Tuning Options. Sophisticated designers should be given techniques for increasing the performance of working applications. For tuning reasons a designer may wish to change the policies for caching the states of selected GemStone objects in Smalltalk's object memory. Also, because GemStone and Smalltalk-80 both provide an environment for method execution, a designer should be given control of where a method is executed. However, designers need reasonable defaults for such choices to allow them to ignore these concerns in the initial implementation stages. Note that this tuning goal conflicts with the object transparency goal.

Transparent Garbage Collection. The Smalltalk-80 environment frees application developers from explicitly deallocating the space occupied by unreferenced

objects. An integrated system should continue to perform this function in an invisible fashion.

Session and Transaction Control. GemStone is a transaction-based database system. Thus developers need control of transactions (atomic actions) and control over the GemStone session from within the Smalltalk-80 environment. In addition, the integrated system should automatically synchronize GemStone object states cached in the Smalltalk-80 object memory with their states within GemStone when a transaction commits or aborts.

Transparent Exception Handling. The Smalltalk-80 environment provides several ways of handling exceptional conditions. The application developer should see a uniform exception handling mechanism independent of the machine where the exception occurs.

Uniform Name Binding. The Smalltalk-80 system binds names to Smalltalk objects at several distinct times: during compilation, during method initiation, and during method execution. Persistent GemStone objects should also participate in each of these binding times. For instance, we noted before that GemStone maintains a `symbolList` of symbolic names on behalf of each GemStone user. Those names should be available whenever name binding occurs in the Smalltalk-80 environment.

3.2 A Design Overview

Our design for integrating the Smalltalk-80 environment with a GemStone object server is based on a new Smalltalk class called `Agent`. An agent is often defined as someone authorized to act in another's interest. The other party is sometimes called the principal. In our design, instances of `Agent` (always Smalltalk objects) act in the interest of principals, which are GemStone objects. An agent knows the GemStone Oop of the principal it represents. Messages sent to a GemStone object from within the Smalltalk environment are directed through the object's Smalltalk agent. `Agent` is an abstract class (only its subclasses have instances) having the subclasses `Proxy` and `Deputy` (Figure 4). Although both *proxy* and *deputy* seem to be synonyms for *agent*, their connotations differ. Proxies usually act on behalf of their principal in a very constrained and limited capacity, such as voting shares of stock or standing in for a bride or groom at a marriage ceremony where the principal cannot be present. In contrast, deputies often have much more authority to make decisions on behalf of their principal. For example, a deputy foreign minister may be authorized to negotiate a trade agreement with perhaps only the final terms subject to ratification by the principal.

Instances of our `Proxy` class act only as forwarding agents, keeping no local information about their principal, other than its GemStone Oops and GemStone class. Each `Proxy` instance forwards messages that it receives to its principal for execution on GemStone and routes the result back to the sender of the Smalltalk message. These proxies offer Smalltalk programmers a substantial improvement over the PIM style of interface because they package messages and arguments in proper form for network communication, unpackage the results, and perform simple translations between Smalltalk and GemStone messages. Such a generic proxy performs correctly with a principal of any GemStone class. Using a minor variant of the proxy scheme, Servio has reimplemented many of

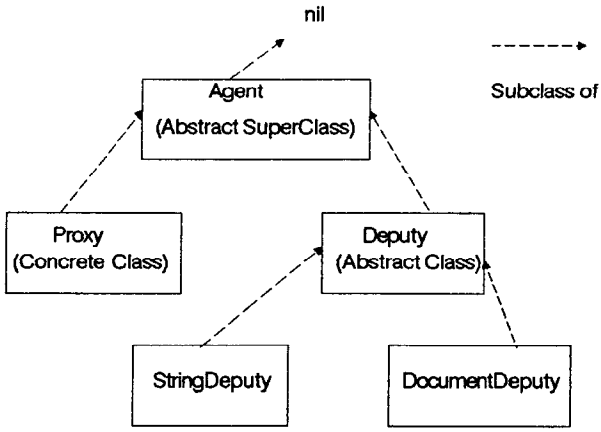

Fig. 4. Relationship among kinds of agent classes.

the OPE tools for the Smalltalk environment, but with much less programming than the original effort, which used C, Windows, and the PIM.

Deputies implement more sophisticated policies for message handling. Deputies may cache all or part of their principal's state in the Smalltalk object memory. Thus different subclasses of `Deputy` can implement distinct policies for message processing and cache management. Some may decide always to cache a GemStone object's state in Smalltalk's object memory and execute messages locally in the Smalltalk environment. Others may selectively execute some messages in the Smalltalk environment and forward others for execution on GemStone. We consider the implementation options for `Deputy` subclasses in more detail later in this paper.

The last new Smalltalk class in our design is `GemSession` (Figure 5), whose instances represent active database sessions with GemStone. An instance of `GemSession` provides all the functions of the PIM. However, only proxies and deputies will use those functions listed in Tables II and III. Most applications will send messages directly to a GemSession only for session and transaction control (i.e., PIM functions in Table I) and for error control. An application may register an error block with a `GemSession` to be executed in the Smalltalk environment with the appropriate error code whenever GemStone raises an exception. A `GemSession` ensures that each GemStone principal represented by a Smalltalk object has at most one agent, via a list called `registeredAgents` that associates each Smalltalk agent with its principal. Before a new agent is created for a GemStone object, `GemSession` consults `registeredAgents` to see whether an agent for that object already exists. This list also supports cache consistency for deputies. When a `GemSession` receives a commit message, it notifies all deputies of its intent to commit. Each deputy then flushes any modified cached state to GemStone. A `GemSession` also notifies deputies whenever a transaction aborts, telling each deputy to invalidate its cached GemStone object state.

Two new Smalltalk objects complete this design. The first is called the `importEquivalents`. This set maps GemStone classes to their equivalents in

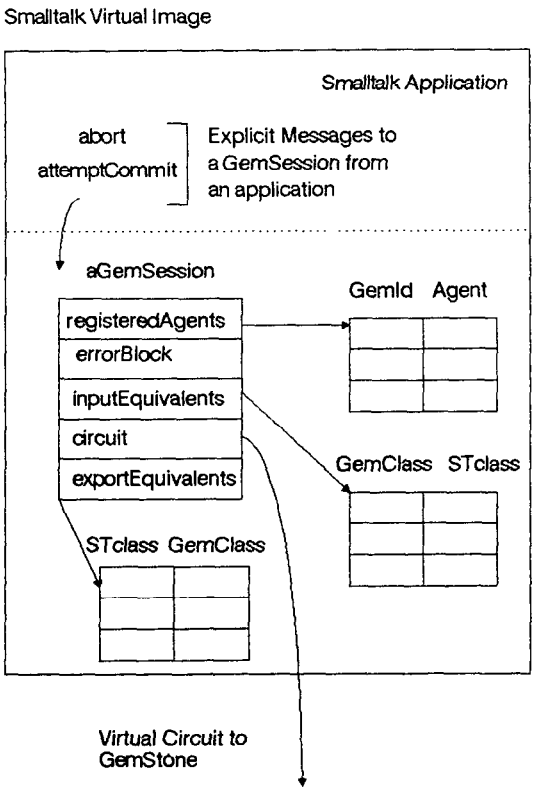

Fig. 5. A GemSession object details.

the Smalltalk environment, typically a deputy or proxy class. GemSession uses `importEquivalents` to decide what kind of a Smalltalk object to create when a GemStone object is first referenced from the Smalltalk environment. For immutable GemStone objects that have well-known behavior, the equivalent import class need not be a subclass of `Agent`. For example, in the cases of numbers and characters, GemSession returns a reference to an equivalent Smalltalk object.

The second new Smalltalk object is called `exportEquivalents`. This set maps Smalltalk classes to their equivalents in the GemStone environment. GemSession uses this set to decide what kind of GemStone object to create when a new Smalltalk object is first exported to GemStone. Ideally, if a Smalltalk object does not already have an equivalent GemStone class in the `exportEquivalents` set, GemSession will create a new GemStone class of the same name, recompile its methods in the GemStone environment, update `exportEquivalents`, and add a proxy class for the new GemStone class to `importEquivalents`. This new proxy class simply forwards all messages. Most Smalltalk methods require only slight syntactic modification before recompilation in the GemStone environment.

The preceding design meets the goals of object transparency, automatic database object creation, tuning options, and session and transaction control. Existing

Smalltalk applications can use agents without noticing that they are such; agents present a protocol much as any other Smalltalk object, yet, they are actually manipulating persistent, shared GemStone objects. Designers can experiment with new caching policies without modifying the application by creating new deputies and modifying `importEquivalents`.

3.3 Implementation of Agents

In an agent's masquerade, it forwards to its principal on GemStone most of the messages that it receives. For efficiency reasons, however, agents (and especially deputies) execute some messages in the Smalltalk environment, particularly those that depend on the principal's identity or class. Selectors like = = (test of identity) and `isNil` (test for nil) are among the many that all agents can process locally.

An Agent must first examine a message intended for its principal before deciding whether to invoke a local Smalltalk method or forward the message for execution on GemStone. Because a class usually inherits messages from all its superclasses (in the agent's case, `Object`), this examination is not always easy. Methods inherited from `Object` do not provide an opportunity for the agent object to decide where work is to be performed. Unfortunately, all sorts of behavior is implemented in `Object` that, in the proxy's case, should be forwarded to GemStone. To handle these inherited messages properly, agents could reimplement all messages inherited from `Object` and forward selected messages on to GemStone.

The scheme above will not work for a generic proxy that forwards any arbitrary message to its principal. This case can be managed if we alter the standard message lookup behavior by putting `nil` in the agent's `superClass` variable and rewrite the instance method `doesNotUnderstand:`. Thus, when a message is sent to an agent, no instance method corresponding to the message is found. Smalltalk in this case sends the `doesNotUnderstand:` message to the original receiver and passes the original message as its argument. The new `doesNotUnderstand:` can establish a default behavior for all messages an agent receives (e.g., always forwarded in the case of a proxy). This same technique can be used to forward messages not reimplemented for each deputy.

Whenever GemStone returns one of its object's Oops to the Smalltalk environment, GemSession ensures that an agent exists for that object by performing the following steps:

(1) See whether this GemStone principal already has an agent by looking in `registeredAgents`. If so, return that agent. Some immutable objects such as `Number` and `Character` instances do not need agents. For such instances, return their Smalltalk equivalents.

(2) Otherwise, create a new agent for the GemStone object. Look in the `importEquivalents` to find which kind of agent to create. If no agent equivalent is found, use a generic proxy that forwards all messages for execution in the GemStone environment.

(3) Add the new agent to the `registeredAgents` list.

Each new subclass of `Agent` must respond to the class message `import: aGemOop`, which should create and initialize a new agent instance for the principal with the given Oop.

3.4 Deputy Implementation Criteria

Designers tuning an application by creating deputies with custom behavior for selected class instances should first examine how those database objects are being used in the application. Designers should consider the following aspects of object usage:

Object Size. Smalltalk does not do well with big objects. Thus, creating a deputy that completely caches a big object's state in the Smalltalk object memory may not make sense. Deputies for such cases may cache fragments of the principal. However, this fragmentation increases the complexity of local methods.

Relative Immutability. Rarely modified objects are easier to manage than volatile ones. In order to manage caches properly, all deputy methods executed locally must mark the cached object appropriately if they modify the cache. This marking can be computationally expensive. Objects known to be immutable can be managed in special ways. For example, `Symbols` in GemStone can be imported directly as `Symbols` in Smalltalk without the need for an agent because they are never changed; their behavior is constant across environments.

Complexity of Behavior. Simple objects with complex methods (e.g., strings) are often worth caching because of the faster execution of methods in the Smalltalk environment.

Interobject Connectivity. Sets of objects that are highly connected may be inefficient to cache in the Smalltalk environment because of the time required to create a proxy or deputy for each member of the set.

Desired Transaction Rate. One difficulty of having many objects with state cached in Smalltalk is that all the dirty ones must be flushed prior to committing a transaction.

Mixing Forwarding with Local Execution. Applications must do so carefully. If a deputy decides to forward a message to GemStone, it cannot know whether the GemStone method depends on another GemStone object that might also have a Smalltalk deputy with a dirty cache. Also, when a forwarded message returns from executing on GemStone, the deputy has no way of determining whether any GemStone objects have been modified as a side effect. One of these modified GemStone objects might have a deputy with an outdated cache.

Bandwidth between Machines. If the channel connecting GemStone with Smalltalk has low bandwidth, then it may be practical to transfer only a few objects between the two environments.

3.5 Deputy Strategies

When designers create new deputy classes, they need to be aware of the interdependence of cache management, transaction management, and message management. This section discusses the options available for each of these, and how the various options interact.

Cache Management. Each deputy has the option of caching a GemStone object's state in the Smalltalk object memory. The designer has several choices on how to manage these caches:

Transitive agent creation. When a deputy caches the state of a GemStone pointer object, it can also create an agent for each object transitively referenced by the cached state. In our example application, when a document deputy is created, both the header and the body can also be converted to deputies, each with its own cached state.

Leaves. A large object-oriented memory (LOOM) leaf [12] can be seen as a generic agent that knows nothing about its principal other than its identifier. Such an agent can delay creating the correct agent until it intercepts its first message meant for its principal. In our example application each entry in a `SetOfEntries` could be treated this way.

Partially cached state. For large objects, a deputy can cache a fragment of the GemStone object. In the example application a document body could be cached in relatively small fragments.

Delayed agent creation. A deputy can always delay creating an agent for part of its internal state if a method that modifies the receiver is not forwarded and operates on the cache. Thus a cache of a GemStone object will reference a Smalltalk object. These references must be changed to agent references, and the referenced objects must be converted to GemStone objects when the cache is flushed to GemStone.

Transaction Management. Deputy caches must be synchronized with their principal's state prior to committing a transaction. This implies that each Smalltalk object referenced from a deputy cache must be exported to GemStone before a commit begins. In our archive example, the application creates a Smalltalk document with a body and header and then adds that document to a `setOfEntries`. The document, its body, and its header must all be converted to GemStone objects because they are transitively reachable from the `SetOfEntries`.

When an application attempts to commit a transaction, GemSession transitively traverses the deputy's cached state, converting all referenced objects to GemStone objects. Then, GemSession flushes dirty deputy caches to GemStone.

Message Management. A proxy that always forwards messages to GemStone never has a problem with cache management. A deputy that caches its principal's GemStone state in the Smalltalk object memory has several options with regard to how it handles messages. For messages it chooses to support with local methods, we see these basic strategies:

Write-through. Perform the method in Smalltalk on the cache, then transmit any changes in the cache to GemStone prior to returing to the sender. Alternatively, the deputy could delay the write-through until just prior to a commit.

Read-back. For all methods that change a cache, a deputy could perform the work on GemStone, then refresh the cached state after the forwarded message returns to the deputy. Thus all reading messages operating in Smalltalk would see the correct state of the cache.

Send-through. Perform the work in both places; that is, both forward the message for execution on GemStone and perform an equivalent method locally. Send-through is an attractive alternative when a single message can cause large changes in a single object's state, such as a global replacement of a substring in a string.

4. CONCLUSION

In this section we evaluate the success of the agent framework, discuss some open research issues with respect to the presented design, and briefly discuss the status of the integration project.

4.1 Summary

The design presented in this paper succeeds at meeting many of the goals for a seamless integration of GemStone with Smalltalk, especially if an application can live with the default behavior of proxies. For those designers not content with the efficiency of the resulting application, this design provides a reasonable factoring to allow incremental tuning by creation of custom deputies. The deputy model allows easy experimentation of alternative cache management strategies. We suspect that once the major classes supplied with GemStone have pretuned Smalltalk deputies, this custom tuning process should not be a difficult chore.

4.2 Open Research Issues

We see the following areas in need of further research before the Smalltalk and GemStone environments behave well as one seamless system:

Unification of Object Models. The Smalltalk and GemStone data models differ in three significant areas: object formats, method execution environments, and language features. GemStone supports objects significantly larger than what most Smalltalk implementations can accommodate and stores large unordered collections (such as `Set` and `Bag`) in a format quite different from Smalltalk. Unlike Smalltalk, GemStone users can declare the type of instance variables in a class definition and declare the type of a collection's elements. Classes provided in the Smalltalk virtual image and in the GemStone "initial" database that share the same name do not necessarily share the same behavior or format. For example, although both environments have an `Array` class, GemStone arrays grow dynamically, whereas Smalltalk arrays grow by using `become:`. Also, GemStone does not implement the `become:` message but does allow an instance to change its class. Further, each system supplies classes not supplied by the other. The process of automatically converting a Smalltalk class to its equivalent GemStone class is complex because of these differences.

Smalltalk Snapshots. The Smalltalk snapshot mechanism is at odds with GemStone's session and transaction control. If a user creates a snapshot of a Smalltalk virtual image while a GemStone session is active or while a transaction remains uncommitted, that snapshot can be inconsistent with the state of the GemStone database when the snapshot restarts. An integrated system should discard the snapshot notion entirely and replace it by a stable object in GemStone that describes the state of the workstation environment. This state information can then be read when the Smalltalk workstation loads the virtual machine. If

this state is stored centrally, then the user can resume work previously suspended on a different workstation.

Transaction Transparency. If one transaction strategy has been shared by a group of applications, then certain deputies or groups of deputies can implement transaction control directly, removing some burden from the application designer.

Symmetric Interface. The interface between the GemStone and Smalltalk environments is asymmetric. Requests for activity always arise from outside the GemStone environment. The two execution environments are in a master–slave relationship, with GemStone being the slave. GemStone objects do not "know" about external environments, nor can they initiate communication with external objects. This asymmetry limits the strategies available for synchronizing state between Smalltalk and GemStone objects, particularly after a message is forwarded to GemStone.

Dirty Cache Identification. Unlike most primary memory subsystems, the Smalltalk-80 object memory does not provide a "changed bit" for identifying modified memory segments. Management of modified caches is very inefficient without such an indicator.

Better Cache Management Policies. If one can modify the Smalltalk-80 virtual machine, more intelligent cache management strategies might be attempted. For example, some multilevel memory schemes (e.g., virtual memory caches [2]) keep the lower layers updated by "writing through" the faster higher layers. These "write through" operations can be asynchronous with the main line process. If we apply this technique to the situation at hand, we reduce the time taken to flush dirty proxies prior to forwarding messages for GemStone execution, by exploiting the workstation's unused CPU cycles during the transacton.

Better Exception Handling. The existing GemStone does not support processes and semaphores in the general manner of Smalltalk. Thus the metaphors for exception handling in the Smalltalk-80 virtual image cannot be used on the GemStone side of an integrated system.

Integrated Development Environment. In order to create a seamless development environment, all the development tools provided in the standard Smalltalk-80 environment must be modified. These tools must be changed because none of them currently has the appropriate menus for handling transaction committing and aborting. Also, each of these tools (browser, inspector, debugger, and workspace) is written to take advantage of only one name dictionary—**Smalltalk.**

Integrated Name Binding. The opportunities for name binding are different between the GemStone and Smalltalk-80 environments. Most applications will want to present the richer GemStone naming environment to their users. To implement this change, the Smalltalk-80 compiler and the mechanism for creating and resuming snapshots must be modified. Also, the objects named in the Smalltalk virtual image must be merged with the objects named in GemStone.

4.3 Project Status

Servio currently has a working version of the Smalltalk–GemStone integration. The current version reflects the design as presented in [11]. Another implementation along the lines suggested by this paper is in progress.

ACKNOWLEDGMENTS

The authors wish to thank Paul McCullough, Allen Otis, Mun Tuck Yap, Tom Ryan of Hewlett-Packard Laboratories, and the reviewers for helping improve the style and presentation of this paper.

REFERENCES

1. DECOUCHANT, D. Design of a distributed object manager for the smalltalk-80 system. *ACM SIGPLAN Not. 21*, 11 (Nov. 1986), 444–450.
2. DENNING, P. J. Virtual memory. *ACM Comput. Surv. 2*, 3 (September 1970), 153–189.
3. GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.
4. GOLDBERG, A. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Mass., 1984.
5. IBM CORPORATION. IMS-VS general information manual. GH20-1260, IBM Corp., White Plains, N.Y., Apr. 1974.
6. KUNG, H. T., AND ROBINSON, J. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.
7. MAIER, D., OTIS, A., AND PURDY, A. Object-oriented database development at Servio Logic. *Database Eng. 8*, 4 (Dec. 1985), 58–65.
8. MAIER, D., STEIN, J., OTIS, A., AND PURDY, A. Development of an object-oriented DBMS. *ACM SIGPLAN Not. 21*, 11 (Nov. 1986), 472–482.
9. MAIER, D., AND STEIN, J. Indexing in an object-oriented DBMS. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Management Systems* (Asilomar, Calif., Sept.) IEEE Computer Society Press, Washington, D.C., 1986, pp. 171–182.
10. PASCO, G. A. Encapsulators: A new software paradigm in Smalltalk-80. *ACM SIGPLAN Not. 21*, 11 (Nov. 1986), 341–346.
11. SCHUCHARDT, B. GemStone to Smalltalk interface. From Poster Session of ACM OOPSLA-86 Conference.
12. STAMOS, J. W. A large object-oriented virtual memory: Grouping strategies, measurements, and performance. Res. Rep. SCG-82-2, Xerox Palo Alto Research Center, Palo Alto, Calif., May 1982.
13. STONEBRAKER, M., Ed. *The Ingres Papers: Anatomy of a Relational Database System.* Addison-Wesley, Reading, Mass., 1986.
14. WHITE, J. E. A high-level framework for network-based resource sharing. In *Proceedings of the National Computer Conference* (New York, N.Y., June). AFIPS Press, Reston, Va., 1986, pp. 561–570.
15. WOO, C. C., AND LOCHOVSKY, F. H. An object-based approach to modeling office work. *Database Eng. 8*, 4 (Dec. 1985), 14–22.

Received July 1986; revised November 1986; accepted December 1986