PC++: An Object-Oriented Database System for C++ Applications

Tin A. Nguyen

Objectivity, Inc.
800 El Camino Real
Menlo Park, CA 94025 USA
tin@objy.com

Mark Wagner, Brent Hoffman

proCASE Corporation
3130 De La Cruz Blvd.
Santa Clara, CA 95054 USA

ABSTRACT

This paper describes PC++ (Persistent C++), an object-oriented database system that supports persistent storage, retrieval, and manipulation of C++ objects by multiple C++ applications executing concurrently on a network. PC++ supports persistent objects by extending the C++ programming language using the C++ inheritance mechanism. PC++ provides object-oriented programming interface, unique object identifier generation, efficient management of object storage and retrieval, optimistic concurrency control, and crash recovery. PC++ also supports long transaction, object versioning, and object clustering which relational database management systems do not support.

PC++ employs a distributed client-server architecture; it consists of an application workspace manager and a database server. The application workspace manager implements the programming interface, performs identifier-to-object mapping, manages the persistent heap space, and performs object clustering. The database server manages files storing objects, creates and reconstructs object versions, and performs automatic crash detection and recovery. The application workspace manager and the database server rendezvous to perform unique object identifier generation, concurrency control, and long transaction processing.

A C++ application integrates with the application workspace manager and executes as a database client process. Clients and the database server communicate using stream sockets. PC++ runs on networked Sun, DEC, and Apollo workstations under the UNIX operating system.

1.0 Introduction

The C++ programming language extends the C programming language to support object-oriented programming [Str86, Ker78]. Many engineering applications (such as CAD/CAM, programming development environments, CASE, AI, and multimedia) are using C++ as the implementation language. These applications require their objects to be persistent— the life time of objects are longer than the duration of program executions.

In engineering applications, objects usually interconnect through references— pointers to other objects. For example, a programming tool may use a syntax tree to represent a program. The syntax tree consists of nodes implemented as objects connected through pointers (right sibling pointers, or parent pointers). Some classes of engineering applications manipulate objects by frequently traversing the references. The references traversal behavior is referred to as *navigation-based referencing* which is distinctive from *value-based referencing* where an object (data record) is accessed based on some indexed attribute. Management information systems are known for having value-based referencing characteristics and are being well supported by relational database management system (RDBMS).

Current RDBMSs do not support ease of modeling complex objects, object versioning, and long transaction [Sto86]. Furthermore, RDBMSs cannot support performance requirements of navigation-based referencing applications (such as CAD applications[Has82]) due to the overhead in transforming the data-independent relational representation from and to complex object representations. With the current workstations, an application can potentially traverse hundreds of thousands to millions of objects per second. Object-oriented database systems (OODBMSs) can achieve high performance when they store and retrieve objects on disk in the format that is very close to the in-memory format. Furthermore, OODBMSs support modeling of complex objects, object versioning, and long transaction.

The goal in developing PC++ is to provide a high-performance object-oriented database system for storing, retrieving, and manipulating persistent C++ objects. Specifically, PC++ is used to construct an advanced integrated programming environment called the *SMARTsystem* [Pro89]. The SMARTsystem (Software Maintenance Analysis and Re-engineering Tools system) consists of an integrated collection of tools for developing and maintaining large C programs. High database system performance is necessary in making the SMARTsystem usable.

DATABASE SYSTEMS FOR ADVANCED APPLICATIONS '91
Ed. A. Makinouchi

PC++ provides some major features that are not supported by relational database management systems such as support for C++ object modeling, long transaction, and object versioning. Section of this paper describes the PC++ system in detail. Section 3 relates PC++ to other works, and discusses future extensions of several PC++ facilities. Section 4 states some conclusions.

2.0 PC++ Architecture and Functionalities

Figure 1 depicts the PC++ system architecture. PC++ employs a distributed client-server architecture that consists of two components: the application workspace manager and the database server. PC++ provides a C++ programming interface through which applications use PC++ facilities. Section 2.1 describes the PC++ programming model. Section 2.2 describes the application workspace manager. Section 2.3 describes the database server. Section 2.4 describes the optimistic concurrency control and crash recovery mechanisms. Section 2.5 reports some performance characteristics of PC++.

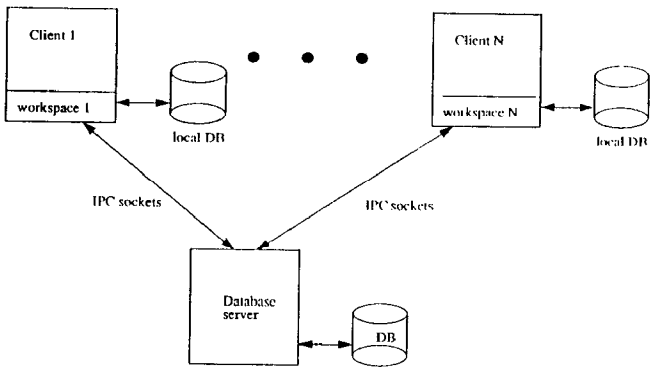

Figure 1. PC++ system architecture

2.1 The Programming Model

A C++ object consists of primitive data, other (nested) objects, and/or pointers to other objects. A persistent C++ object in a PC++ database contains an object identifier uniquely identifying the persistent object. A unique identifier is used as the persistent pointer to a persistent object. A persistent C++ object at all times contains object identifiers instead of pointers to other objects (whether the persistent object is in main memory or on disk).

PC++ provides an object-oriented programming interface consisting of three pre-defined C++ classes (types): PERSISTENT, PERSISTENT_ID, and WORKSPACE. The class PERSISTENT, which all classes of persistent objects must derive (inherit) from, supplies the object identifier and functions for referencing and modifying persistent objects. The class PERSISTENT_ID defines an object identifier and supplies a function for mapping an object's identifier to the object.

The pre-defined class PERSISTENT defines the following properties for persistent objects:

```
class PERSISTENT {
    public:
        PERSISTENT(int object_size,
                cluster c = DEFAULT_CLUSTER);
        virtual ~PERSISTENT();
    protected:
        PERSISTENT *modify();
        <other protected functions>
    private:
        PERSISTENT_ID id;
        <other private data>
};
```

The pre-defined class PERSISTENT_ID defines the following properties for object identifiers:

```
class PERSISTENT_ID {
    public:
        PERSISTENT *operator->();
    protected:
        <protected functions for manipulating ids>
    private:
        <declarations for page and object numbers>
};
```

For a class P to be persistent, an application must define class P deriving (directly or indirectly) from the class PERSISTENT. In addition, the application must define an associated object identifier class (for example, P_ID) deriving from the class PERSISTENT_ID. The object identifier class P_ID serves as the persistent pointer type for its associated persistent class P.

For example, an incremental programming tool might represent a program as an abstract syntax tree [Fis88] and would like the abstract syntax tree to be persistent. The abstract syntax tree consists of a set of nodes containing pointers (left most child, right sibling or parent) to other nodes in the tree. The nodes can be modeled as objects. The abstract syntax tree is made persistent by making its nodes persistent. The programming tool, for example, would define the following classes:

```
class node : public PERSISTENT {
    public:
        node(<args>);
        node_id get_leftmost_child() { return lmc; }
        void set_leftmost_child(node_id lmc_arg)
        { modify(); lmc = lmc_arg; }
        int get_value();
        <other functions specific to node>
    private:
        node_id lmc;
        <other data specific to node>;
};
```

```
class node_id : public PERSISTENT_ID {
    public:
        node(<args>);
        /* pointer-to-identifier conversion function */
        node_id(node *);
        /* node id-to-node mapping operator */
        node *operator->()
        { return ((node *) PERSISTENT_ID::->()); }
};
```

PC++ supports only persistent heap objects; an application cannot create static or local persistent objects. To create a persistent object, an application uses the C++ *new* operator. For example, to create a node:

```
node *n = new node(<args>);
```

In C++, the order of constructors execution for a class hierarchy is from ancestors to descendants. Thus, the constructor of the class PERSISTENT executes *before* the constructor of the class node executes. The PERSISTENT constructor allocates space for a new node object, and assigns a unique identifier to the new node object. The node constructor passes the size of the node information through the *object_size* formal argument of the PERSISTENT constructor. The node constructor also optionally specifies a cluster where the new node object belongs through the *c* formal argument of the PERSISTENT constructor. Clusters are referred to by cluster identifiers (enumeration constants) that are defined in a C++ header file.

When modifying an object, an application must invoke the *modify* function which marks the object as modified so that during transaction processing the application workspace manager can identify which objects need to be sent to the server for updating. For example, the *set_leftmost_child* member function of class node invokes the *modify* function before modifying the *lmc* field.

To delete a persistent object, an application uses the C++ delete operator. The delete operator invokes the PERSISTENT virtual destructor *after* the descendants' virtual destructors. The PERSISTENT virtual destructor immediately reclaims the space and the object identifier of the persistent object being deleted. PC++ employs the C++ heap space management model, and assumes that the application programmer is responsible for assuring that there is no dangling persistent pointer. The PERSISTENT constructor and destructor for persistent C++ objects have the same semantics as those of the C++ *malloc* and *free* library functions, respectively, for non-persistent objects. Thus, PC++ does not employ a garbage collection mechanism (as there is no garbage collection facility in C++).

Pointers to persistent object are automatically converted to persistent object identifiers by conversion functions. Thus, persistent object pointers are used syntactically the same as non-persistent object pointers. For example, the following C++ statements set node n2 as node n1's leftmost child:

```
node *n1 = new node(...);
node *n2 = new node(...);
n1->set_leftmost_child(n2);
```

Note that the function *set_leftmost_child* accepts a node pointer (n2) as its argument because the constructor *node_id(node *)* of the class node_id automatically converts a node pointer to a node identifier. PC++ provides a C++ macro implementing the conversion function to prevent programming error and to relieve implementation burden on the programmer.

The overloaded operator -> defined in each persistent object identifier class allows the natural use of C++ syntax in accessing a persistent object's members through the persistent object's identifier. For example, the following C++ expression invokes the *get_value* member function of node n1's leftmost child which is an object identifier of type node_id:

```
n1->get_leftmost_child()->get_value()
```

The overloaded operator -> of the class PERSISTENT implements the identifier-to-object mapping. Each persistent class needs to provide only the proper casting of the mapping result. For example, the overloaded operator -> of the class node invokes the overloaded operator -> of the class PERSISTENT and then casts the result to become a node pointer.

The pre-defined class WORKSPACE supplies functions for checkpointing, transaction processing, and initialization. The WORKSPACE class supplies the following interface:

```
class WORKSPACE {
    public:
        status checkin();
        status abort();
        status checkpoint(char *checkpoint_file);
        status update(int version_number = LATEST);
        status assign_root(PERSISTENT_ID root_id);
        PERSISTENT *get_root();
        status initialize(char *bootstrap_file);
};
```

PC++ supports long transaction by providing a checkpoint and a checkin function. The checkpoint function saves all new and modified objects to a user-specifiable file in case there is a system failure. The checkin function requests the database server to perform a transaction commit, and sends all new and modified objects to the server for transaction processing. PC++ starts a transaction automatically after a transaction commit or abort. To abort a transaction, an application invokes the abort function which resets the workspace.

A PC++ database is an extension to a C++ application's heap space. The identifier-to-object mapping function provides implicit indexing on object identifiers which is critical for navigation-based applications. Currently, PC++ does not provide any explicit indexing capability for accessing objects based on attributes or names. PC++ assumes that all persistent objects are reachable from a *root* object. The WORKSPACE class provides

the function *assign_root* for assigning an object as the root object, and the function *get_root* for accessing the root object from which all objects can then be accessed through their identifiers.

The WORKSPACE class provides an initialization function for starting up a C++ application. The initialization function accepts a UNIX path to a database bootstrap file. The database bootstrap file contains the location of the followings: the database server, the database files, the checkpoint file, and the local persistent heap swap file. Both the application workspace manager and the database server use the bootstrap file for starting up.

2.2 The Application Workspace Manager

An application integrates with the PC++ application workspace manager and runs as a database client process. The workspace manager implements the PC++ programming interface, manages local persistent objects, and interfaces to the database server through a stream socket interprocess communication channel on behalf of the application. Figure 2 depicts the structure of the application workspace manager.

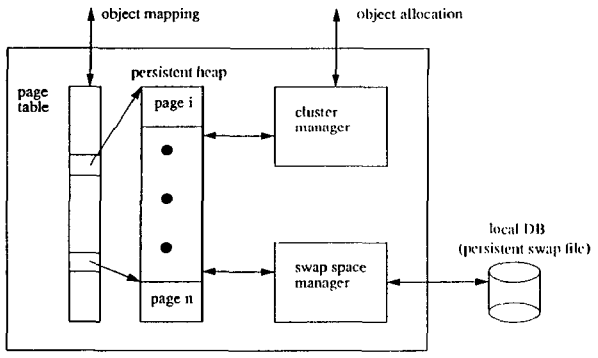

Figure 2. Application workspace manager structure

The application workspace manager consists of a heap space for persistent objects, a variable-size page index table, a persistent swap space manager, and an object clustering manager. The application workspace manager performs identifier-to-object mapping, new object creation, object fetching, and long transaction processing in coordination with the database server.

A PC++ object database consists of a number of 4K-bytes pages. A *page* contains some header information, a variable-size object index table, and a set of objects. A page is the physical unit of fetching. An object identifier consists of a page number and an object number. The page number specifies which page the object is in. The object number specifies which object on the given page.

The PERSISTENT operator -> maps an object identifier to the object. Given an object identifier, the operator -> uses the identifier's page number to look up the page through the page index table. If the page is not in memory, the persistent swap space manager fetches the page into memory. Once the page is in memory, the operator -> uses the identifier's object number to find the object through the object index table on the page. The operator -> returns a pointer to the object.

The persistent heap space consists of pages grouped in clusters. When allocating an object in a specified cluster, the cluster manager finds an existing page with enough free space for the object. If there is not enough free space for the object in the specified cluster, the cluster manager allocates a new page for the cluster. The cluster manager then allocates space to the object on the page, and assigns to the object an identifier whose page number is that of the page and whose object number is the next free index in the page's object index table.

The variable-size page index table is a hash table whose entries are pairs of page numbers and pointers to pages in the heap space. The page index table grows as the cluster manager allocates new pages.

The application workspace manager provides and manages additional swap space for persistent objects because the application's process swap space may exhaust before the application commits a long transaction. The application workspace manager employs a least recently used (LRU) algorithm to swap pages. the algorithm freely discards pages containing only old unmodified objects since these pages can always be refetched from the database. For pages containing modified or new objects, the algorithm swaps them in from and out to the persistent swap file.

2.3 The Database Server

The database server performs page fetching and reconstruction per client requests, database update and object versioning during checkin, crash detection and recovery, and message delivery between clients and an application-specific lock server. Figure 3 depicts the structure of the database server.

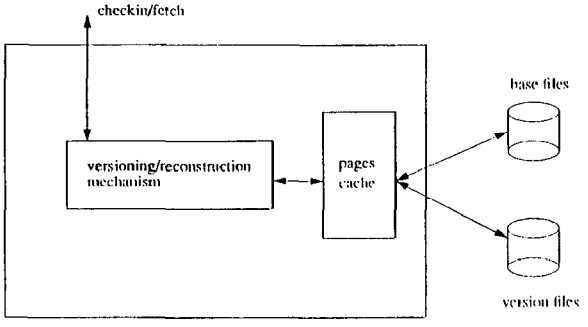

Figure 3. Database server structure

A PC++ object database consists of many database versions. A database version comprises all updated and new objects from a transaction. The versioning mechanism is page-based. Each database page consists of a set of base objects (the latest version) and sets of version objects (previous versions). The base objects of a page reside on a base page that is stored in a base file. Each version object of a page contains the delta objects for a previous version of the page. Version objects reside on version pages stored in version files. A base page and its associated version

pages are logically linked together to facilitate new version creation (during checkin) and page reconstruction (during fetching).

The server employs a reverse delta reconstruction algorithm to reconstruct any version of a database. When a client requests a page of a previous database version, the server applies the version objects of a page to the base page. Since multiple clients may request the same page of different versions, the server caches both base and version pages. The server uses an LRU algorithm for replacing pages in the page cache.

2.4 Concurrency Control and Crash Recovery

PC++ provides optimistic concurrency— concurrent object creation, access, and modifications by multiple clients— through mechanisms for unique object identifier generation, and workspace version updating and conflict resolution. By providing optimistic concurrency, PC++ does not need to provide a locking facility.

The application workspace manager generates unique identifiers by requesting the database server to dynamically allocate unique sets of contiguous identifiers on demand. When a client creates a new object and exhausted the previously allocated set of unique identifiers, the client sends a message to the server requesting for a new set of unique identifiers. The server returns a set of unique page numbers that the client can freely use to allocate new objects.

PC++ enables a client to validate the effects and resolve conflicts (if there are any) resulting from other clients' checkins by providing an update workspace function where all the old objects at version V in the client's workspace are replaced by their latest version. When a client successfully checkins a new database version, other clients are neither notified nor updated immediately. Consequently, a client who starts out at version V of the database may not always able to checkin on top of version V. And therefore, the client needs to use the update workspace function to validate the semantic correctness of the new database version before the client commits its checkin.

PC++ automatically detects failures and recover through a rollback mechanism. If there is a failure during checkin, the server rollbacks to the latest consistent version before the checkin started. Rollback consists looking for base pages with the new version number, and reconstructing them back to the previous consistent version. Checkin processing facilitates rollback by first constructing the version object (for the previous version) of a page and writing the version object out to disk before overwriting the base page with the new version. Thus, the server can always reconstruct the previous version of a base page.

2.5 Performance Characteristics

PC++ serves as the foundation for the SMARTsystem. PC++ performance is characterized by numerous SMARTsystems benchmarks that produce databases ranging in sizes from hundreds of kilobytes to hundreds of megabytes. These benchmarks produce persistent C++ objects with average size of 30 bytes.

Each SMARTsystem benchmark consists of invoking a series of SMARTsystem tool commands simulating code development and maintenance activities (text navigation, editing, parsing, symbols creation and lookup, and call graph browsing) on a production C program. The benchmark showed that PC++ takes 15 microseconds to map an object identifier to the object on a Sun 3/60 machine; this translates to roughly 66,000 object traversals per seconds (providing that they're already in memory). The benchmark also showed that PC++ takes 100 microseconds to create a new object (roughly 10,000 new objects per second).

The SMARTsystem performs well when the working set [Den68] of a benchmark fits the available main memory. For example, navigating and editing with the SMARTsystem syntax-sensitive editor is as fast as using Emacs[Sta81]. However, some incremental global symbol generation appears sluggish because it traverse more objects than the available memory can hold. Profiling the later case showed that the system spent 90% of its time waiting in the UNIX read system call [Rit74], and the remaining 10% of its time in executing tool functions, other system calls, or the identifier-to-object mapping function.

Heuristically grouping objects (for example, objects belonging to the parse tree, or the symbol table) using the static clustering mechanism improved SMARTsystem performance from 10% to 20%. The improvement comes from increased locality, and less internal fragmentation because objects of the same type (and thus same size) are grouped together. On the average, 6% of a page's space is overhead (header information and object index table), and less than 2% of a page's space is unoccupied by objects.

3.0 Related and Future Works

The PC++ programming model is similar to that of PS-Algol [Atk83]. Both systems view the persistent heap space as part of virtual memory, and assume that persistent objects are reachable from a root object. PC++ applications, like the E system [Ric89] applications, use C++ to define both persistent types and their methods. Thus, PC++ and E smoothly integrate their object-oriented database systems with the C++ programming language. Other systems such as the Vbase system [And87] provide one language for defining types and a host language (some superset of the C programming language) for defining methods. Thus, Vbase provide more flexibility for supporting other host languages.

Some SMARTsystem tools produce intermediate objects in computing some final results. After obtaining the final results, there is no need to save the intermediate objects to the database. By making persistency independent from type, PC++ can allow applications to specify whether an object of a class deriving from the class PERSISTENT is persistent. Specification of persistency can be done at object creation time through an argument of the *new* operator of the class PERSISTENT [Str90]. PS-Algol and the ODE system [Arg89] provide strict orthogonal persistence.

Like in Avalon/C++ [Det88], in PC++ persistence types are defined by inheriting from a pre-defined type that supplies persistent properties. PC++ does not employ compilation technology, such as employed by the E system, to support persistency.

An object identifier in PC++ is a physical address. Thus, an object's physical location (the page location) can be computed from its identifier without using a lookup table. Consequently, the identifier-to-object mapping mechanism is simple and highly efficient. The object size limitation does not hamper the construction of the SMARTsystem in any way. However, applications in image processing, graphics, or multimedia may require support for large objects with arbitrary sizes. PC++ can support arbitrarily-sized objects by dividing the object name space into two name spaces: one for small objects (less than 4K bytes) and one for larger objects.

A more flexible object clustering mechanism should include dynamic clustering. With dynamic clustering, the application workspace manager creates new clusters at run time on demands by the application. The existing unique object identifier generation mechanism can be generalized for generating dynamic cluster identifiers. With both static and dynamic clustering, PC++ can support all object clustering options as described in [Kim90].

4.0 Conclusions

PC++ is a high performance object-oriented database system for constructing C++ applications. The design of PC++ fully exploits the capability of a modern workstation so that C++ applications whose working sets match the workstation's available main memory can perform well. By focusing on and achieving high performance, PC++ is commercially usable.

PC++ supports persistent C++ objects by extending C++ using the C++ inheritance mechanism. PC++ achieves acceptability and flexibility by adhering to and implementing the C++ object-oriented programming and data model, and by providing a simple object-oriented programming interface. PC++ is acceptable to C++ programmers because it does not require significant additional training. PC++ is flexible for C++ programmers because it allows them to naturally extend the C++ programming model to include persistency.

5.0 Acknowledgments

The authors acknowledge design and implementation contributions from Jerry Barenholtz, Robert Evans, Brian Gill-Price, Efrem Lipkin, Timothy Rentsch, and Anatol Zolotusky. Susan Najour and Lynn Rohrer improved this paper's presentation.

6.0 References

[Arg89] R. Agrawal, and N. H. Gehani, "ODE (Object Database and Environment): The Language and the Data Model", *Proceedings of the 1989 ACM-SIGMOD International Conference on the Management of Data*, Portland, Oregon, June 1989.

[And87] T. Andrews, and C. Harris, "Combining Language and Database Advances in an Object-oriented Development Environment", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Orlando, FL, October 1987.

[Atk83] M. P. Atkinson, P. J. Bailey, W. P. Cockshott, K. J. Chisholm, and R. Morrison, "An Approach to Persistent Programming", *Computer Journal*, 26(4), 360-365, 1983.

[Den68] P. Denning, "The Working Set Model for Program Behavior", *Communications of the ACM*, 11(5), 323-333, May 1968.

[Det88] D. D. Detlefs, M. P. Herlihy, and J. M. Wing, "Inheritance of Synchronization and Recovery Properties in Avalon/C++", *IEEE Computer*, 21(12), 57-69, December 1988.

[Fis88] C. N. Fischer, R. J. LeBlanc, Jr., *Crafting a Compiler*, Benjamin/Cummings, 1988.

[Has82] R. Haskings, and R. Lorie, "On Extending Functions of a Relational Database System", *Proceedings of the 1982 ACM-SIGMOD International Conference on the Management of Data*, Orlando, FL, June 1982.

[Ker78] B. W. Kernighan, and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[Kim90] W. Kim, "Architectural Issues in Object-Oriented Databases", *Journal of Object-Oriented Programming*, 2(6), 29-38, March 1990.

[Pro89] proCASE Corporation, *The SMARTsystem Reference Manual*, 1989.

[Ric89] J. E. Richardson, and M. J. Carey, "Persistence in the E Language: Issues and Implementation", *Software-Practice and Experience*, 19(12), December 1989.

[Rit74] D. Ritchie, and K. Thompson, "The UNIX Time-sharing System", *Communications of the ACM*, 17(7), 365-375, July 1974.

[Sta81] R. Stallman, "EMACS: The Extensible, Customizable Self-Documenting Display Editor", *Proceedings of the 1981 ACM-SIGPLAN-SIGOA Symposium on Text Manipulation*, June 1981.

[Sto86] M. Stonebraker ed., *The INGRES Papers: Anatomy of a Relational Database System*, Addision-Wesley, 1986.

[Str86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[Str90] B. Stroustrup, *The C++ Programming Language — Annotated Reference*, Addison-Wesley, 1990.

Sun 3/60 is a trademark of Sun Microsystems, Inc.
DEC is a trademark of Digital Equipment Corporation
Apollo is a trademark of Hewlett-Packard Corporation
UNIX is a trademark of AT&T Bell Laboratories
SMARTsystem is a trademark of proCASE Corporation