Constructing A Distributed Object-Oriented System with Logical Constraints for Fluorescence-Activated Cell Sorting

Toshiyuki Matsushima
Herzenberg Laboratory, Genetics Department,
Stanford University, Stanford, CA 94305
matu@cs.Stanford.EDU

Abstract

This paper [1] describes a fully distributed biological-object system that supports FACS (Fluorescence Activated Cell Sorter)[2] instrumentation. The architecture of the system can be applied to any laboratory automation system that involves distributed instrument control and data management.

All component processes of FACS (such as instrument control, protocol design, data analysis, and data visualization), which may run on different machines, are modeled as cooperatively-working "agents." Communication among agents is performed through shared-objects by triggered methods. This shared-object metaphor encapsulates the details of network programming. The system facilitates the annotation of classes with first-order formulae that express logical constraints on objects; these constraints are automatically maintained upon updates. Also, the shared-object communication and polymorphic triggered methods are exploited to produce a homogeneous interface for instrument control.

Introduction

We are developing a computer support system for fluorescence-activated cell sorters at the shared FACS facility in the Stanford University Medical Center. Through the process of applying modern computer technologies, such as distributed computing, object-oriented data management, and logic-based constraint maintenance, we have acquired a significant amount of experience that can be shared with others who are developing laboratory automation systems. This paper presents our design considerations and decisions, the architecture of our system, some advanced features of the system, and the solutions to some problems we encountered.

[1]This work is supported by NLM grant LM04836 and NIH grant CA42509.

[2]We use "FACS" as the acronym for "Fluorescence Activated Cell Sorter" in this paper.

The paper is organized as follows.

- The second section, **Application Domain**, explains FACS experiments and the requirements for a support system.
- The third section, **Goals and Design Decisions**, discusses the goals of the system, some possible ways of attaining those goals, and our decisions.
- The fourth section, **Technical Features**, describes the architecture of our system and some advanced features, such as logical constraint maintenance and agent communication, in the framework of an object-oriented data management system.
- The fifth section, **Discussion**, describes some benefits that a biologist or a biological laboratory could get from our system architecture. This section also compares our system with other laboratory automation systems.
- The sixth section, **Conclusion**, completes the paper by summarizing our system and mentioning some of our plans for the future.

Application Domain

This section explains fluorescence-activated cell sorting, the application domain for which our system has been developed.

FACS Experiments

The fluorescence-activated cell sorter is a device widely used in immunology and molecular biology to identify protein expressions on cell surface. FACS experiments are performed in these basic steps:

- Preparation
  1. Given cells of interest, we choose a set of reagents (known antibodies attached to specific fluorescent dyes), and mix them in a solution. If a specific protein on the cell surface matches the antibody in a reagent, that reagent will stick to the cell. The amount of protein on the cell surface determines the amount of reagent that will stain the cell. Choosing reagents is a complicated process that requires biological expertise.

2. Adjust the configuration of FACS instrument to match the reagents. For example, the scaling factors for measured parameters must be adjusted to suit particular reagents.

- Cell Analysis

1. The stained cells are conveyed in a microscopically thin fluid flow, then exposed to laser beams of specific wavelengths. By measuring fluorescent light emitted from each cell when it passes each laser beam, we can determine how much of each reagent is bound to that particular cell. The emitted light is measured from several directions through specific band-pass filters, producing multi-dimensional values.
2. The measured values are stored in the computer system in real time. Later, the distribution of the values is displayed by computer in many ways to identify cell subpopulations of biological significance. The subpopulations are represented by subdomains of the value space.

- Cell Sorting
  Once a specific cell subpopulation has been identified by cell analysis, the FACS can sort out the cells of that subpopulation.

1. The subdomain information is loaded to the FACS instrument control system.
2. The stained cells are excited by laser light as in the first step of cell analysis, above.
3. Based on the measurement of emitted light, the system determines whether the cell is in the specified subpopulation.
4. By the same mechanism used in ink-jet printers, the flow is transformed into tiny droplets containing at most one cell each; the orbit of each droplet can be controlled by an electrostatic mechanism. With elaborated synchronization, the system can identify which cell is in which particular droplet. Thus it can direct the orbits of the cells in the subpopulation into a specific tube (**Figure 1**).
5. At the same time, measured values are stored in the computer system for further analysis and verification of the sorting process.

Issues and Requirements

FACS Experiment Design Designing a FACS experiment requires choosing a set of reagents to observe particular properties of a given cell population, which can be a very complicated process. The designer must consider, for example,

- what reagents can identify which cellular property;
- what lasers are installed in the machine, which fluorescent dyes are compatible with each laser wavelength, and which band-pass filters are used;

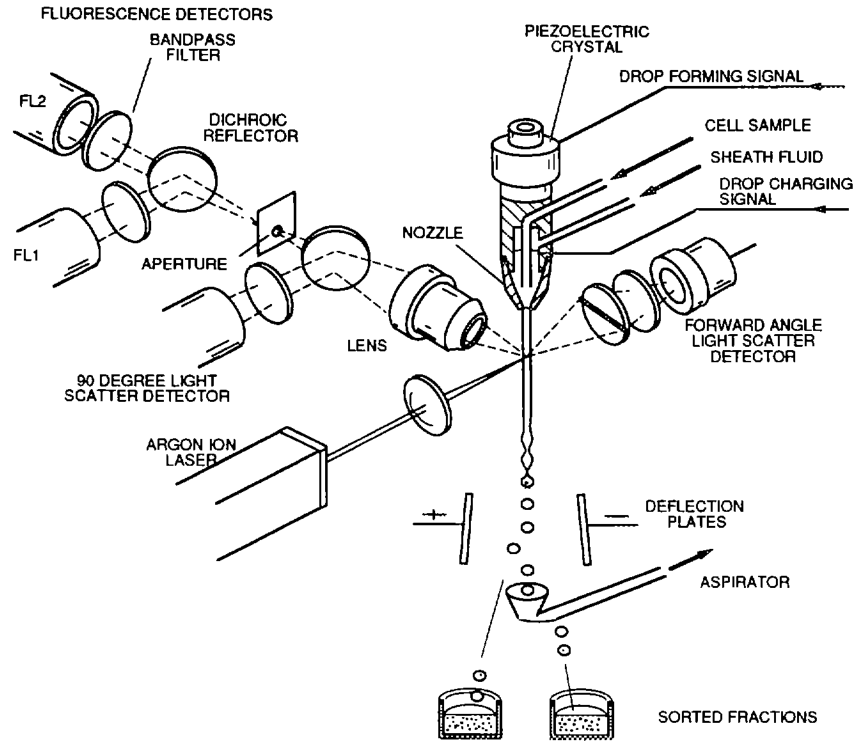

Figure 1. Mechanism of Cell Sorting

- what excitation intensity and spectrum distribution a reagent (a fluorescence dye) has for a given laser wavelength;
- which combination of reagents should be used for a given laser configuration. (For example, it is not useful to mix reagents that have different antibodies conjugated to the same dye.)

A FACS support system should store such knowledge in a knowledgebase, and include logical descriptions to verify the design, e.g. correct reagent use. A tool for experiment design should be able to access the current configuration (lasers, filters, etc.) of the FACS machine.

Instrument Control The FACS instrument should be configured based on the set of reagents (dyes) to be used and measurement criteria such as what parameters to measure. Thus experiment information should be accessible to the machine control module. The correct configuration of the instrument should be described logically and maintained automatically.

Furthermore, we should develop a homogeneous interface for instrument control, so that other components of the system can treat different FACS instruments uniformly as "abstract FACS machines." This is critical, because we operate several FACS instruments of different hardware designs in our laboratory.

Distributed Data Management and Process Control Our FACS instruments are situated in different locations. Experimental protocols, data produced by the experiments, and the instruments themselves must be represented and managed in a distributed environment. Also, the processes should be distributed in order to utilize computer resources effectively. For example, computationally-intensive processes such as data analysis and calculation for data visualization should be done by high-end servers, while actual data display should be done by end-user workstations.

These conditions require a uniform management of distributed data and processes. In order to emphasize the distributed, autonomous and cooperative nature of component processes of the FACS system, we call them *FACS agents*.

Goals and Design Decisions

This section explains our design decision to construct a distributed object-oriented system that has integrated constraint maintenance and communication facilities.

Distributed Object System

As described in the previous section, the FACS domain knowledge (data) should be shared by multiple processes that run on different machines. Thus, a distributed data management system is necessary. Also, in order to provide consistent and secure data management upon simultaneous updates, network failure, and process failure, we have decided to use a full-fledged commercial database management system that has concurrent control and recovery, rather than building everything on top of the file system supplied by an operating system.

Relational Database vs Object-Oriented Database FACS domain objects have complex structures and need to reference one another. Thus, in a process address space, we can naturally represent them as complex objects (e.g. instances of C++ classes).

In order to store such objects in a relational database system, we have to "decompose" objects into flat/hierarchical tables and introduce keys in order to reconstruct references by join operations. Then, when accessing objects in the database, we have to issue a query, load tuples, and reconstruct objects. As a matter of fact, we had been exploring this direction as a part of the Penguin system [Barsalou 1990], [Sujansky et al. 1991], until a couple years ago. Although the idea of establishing a generic way of converting objects to and from a relational database was interesting [Barsalou et al. 1991], it turned out that the overhead of the conversion was too large for our application. In some cases, it took as long as 20 minutes(VAXStation 3000, local disk, local SQL server) to construct objects out of relational tables with a few thousand tuples, when outer-join was required. This problem may be resolved if commercial relational database vendors support outer-join. Yet, it is unlikely that we could get enough performance for interactive use. Penguin is suited only to applications that load all data objects into memory at the beginning of a user session, then store them back at the end of the session.

In an object-oriented database, on the other hand, the persistent images of objects are close to their on-memory images. The references among objects are efficiently converted by the database system with the swizzling operation. Furthermore, the consistency of references is maintained by the database system.

In our application, users should be able to access each particular object interactively. Thus we decided to utilize more efficient object-oriented database.

Agents Communicating via Shared Objects As discussed in earlier sections, FACS agents communicate extensively to perform cooperative work. Thus, providing an abstract and comprehensive programming interface is critical.

Since we adopted the strategy of distributed object-oriented data management, it was natural to take advantage of it by using shared objects for passing information among processes. Basically, agents running on different machines share objects; when an agent issues a triggering method on a shared object, all the agents that are "interested" in the object are notified and the specific methods associated with the trigger are executed automatically in each agent's process context.

With this framework, the design of a communication protocol is equivalent to that of triggered methods. Thus, agents need not be aware of the details of network communication; they simply act on the shared object.

This framework provides a better programming interface than traditional communication metaphors, such as message passing or remote procedure call (RPC).

Message Passing/RPC vs Shared Objects With a message passing scheme, such as one using internet socket, a programmer has to explicitly parse objects out of stream or expand them into stream. With RPC, the programmer need not parse objects. However, in both cases, if an agent wants to know the status of other agents, it must be explicitly included in the communication protocol.[3]

On the other hand, in communication via a shared object, if an agent sets a specific substructure of the shared object expressing that agent's state, it will be automatically visible to other agents. This makes the design of a communication protocol much simpler. Of course, in the lower level hidden from the programmer, extra communication takes place to update the memory image of the shared object in each agent. We think this overhead is negligible, because the communication bandwidth has been growing larger in recent years.

One more advantage of the shared-object scheme is that we can use the "conferencing metaphor" in designing a protocol, rather than peer-to-peer communication or general broadcasting. More specifically, more than two agents can communicate by sharing a specific object in such a way that all of those agents and only those agents can see the updates on the shared object performed by one of them. This also makes the communication protocol simpler.

Built-in Constraints Maintenance

Since we have adopted the object-oriented framework, it is better to implement the constraint maintenance mechanism for each object than to have a separate constraint-maintenance module. Furthermore, describing constraints at the object level has these advantages:

- It is easy to localize constraint maintenance, which increases program efficiency and maintainability.
- When combined with triggered methods of objects, we can activate procedures for "error handling" in multiple agents sharing common objects.

[3]The author had previously designed and implemented a message handler in Penguin system to simulate multi-threaded programming on VAX/VMS, so that each module of the Penguin system could act as "agent." Although it worked fine, it was difficult to design the communication protocol carefully so that the process status of agents could be determined.

As discussed later, our system represents the constraints in first-order formulae with an operational semantics of boolean functions.

Rule-based Constraints vs Function-based Constraints

Traditionally, a rule-based system is used to maintain complex logical constraints. It allows a programmer to express the constraints "declaratively." However, such a system tends to result in an exponentially slow program if the programmer tries to write logically *elegant* rules. Thus, we have to exploit various techniques to reduce search space, e.g. hierarchical organization of rules, etc. In practice, we usually end up exploiting the procedural execution semantics of inference engines to speed up our program; or we use foreign functions (predicates) written in a procedural language. Since the rule-based system is not built for procedural programming style, we are forced to introduce an undesirable kludge.

On the other hand, our approach is to impose procedural/functional semantics to logical formulae from the beginning, while preserving the descriptions themselves as purely logical. Namely, we provide an object-oriented functional language with a construct to express first order formulae as boolean functions; the language has an operational semantics that realizes a correct model of the first-order formulae[4]. This approach provides a seamless integration of logical description and procedural execution. If the performance is not important, as in the prototyping phase of development, a programmer can just express the constraints in first-order formulae constructed from primitive predicates with quantified variables and logical operations. Moreover, we can get benefits from well-established optimization techniques for functional languages and set-oriented languages, such as LISP and SQL. If optimization by the compiler does not provide enough performance, then the programmer can rewrite some "hot spot" first-order formulae using procedural constructs, preserving the semantics of the original formulae.

Technical Features

This section describes the architecture of our system as well as some system design issues.

System Structure

The system consists of the following components (agents) (**Figure 2**).

- Instrument Controller Agent
  Agent to control a FACS instrument through direct connection; each FACS instrument has a dedicated instrument controller agent.

[4]More precisely, *most* of the first-order formulae, because a programmer can write first-order formulae that have non-terminating execution.

- Instrument Console Agent
  GUI (Graphic User Interface) for the instrument controller agent. It shares an *instrument object* with the instrument controller agent. This agent realizes a kind of virtual FACS instrument[Santori, M. 1990] by providing graphical console panels. By virtue of polymorphic triggered methods, we can reuse the same code for controlling FACS instruments of different hardware implementations. We will explain this in detail later.
- Protocol Editor Agent (Experiment Designer)
  GUI for designing FACS experiments. It creates a *protocol object*.
- Data Browser Agent
  GUI for browsing protocol objects and *collected data objects*.
- Data Analyzer Agent (Graph Generator)
  Agent to analyze data objects and generate *graph objects*.
- Data Viewer Agent
  GUI for displaying graph objects; the agent also supports interactive gating (subpopulation specification). The subpopulation specification will be associated with a protocol object to direct sorting. (This is not shown in the figure.)

Logical Constraint Maintenance

This subsection describes the mechanism of the logical constraint maintenance facility of **Quartz** as well as optimization issues.

An Data Description Language: Quartz **Quartz** is a simple object-oriented data description language with a CLOS-like type system. It allows the attachment of a first-order formula to each class in order to express constraints. **Quartz** compiles first-order formulae into boolean functions and procedures, which are used to maintain the constraints automatically; the system tries to keep the objectbase a correct model of the formulae (theory). **Quartz** objects are either entity-objects (object-identifiers) or value objects (integers, reals, strings, or sequences of **Quartz** objects). All entity-objects are persistent, although some entity-objects could reside only on memory for their entire lifetimes. Complex objects are represented as collections of *attribute mappings* that map object-identifiers to **Quartz** objects. The attribute mappings are implemented as associative sets using Btrees, which can be distributed over the network. The formal model of **Quartz** and the details are described in [Matsushima & Wiederhold 1990].

In order to get some idea of the language and its semantics, we consider the following simple example:

```
(defconcept reagent (base entity) (isa top)
  ((antibody string) (fluorochrome string))
   ;; ([attribute name] [attribute value type])
  (restriction true) )
    ;; (restriction [constraints formula])
```

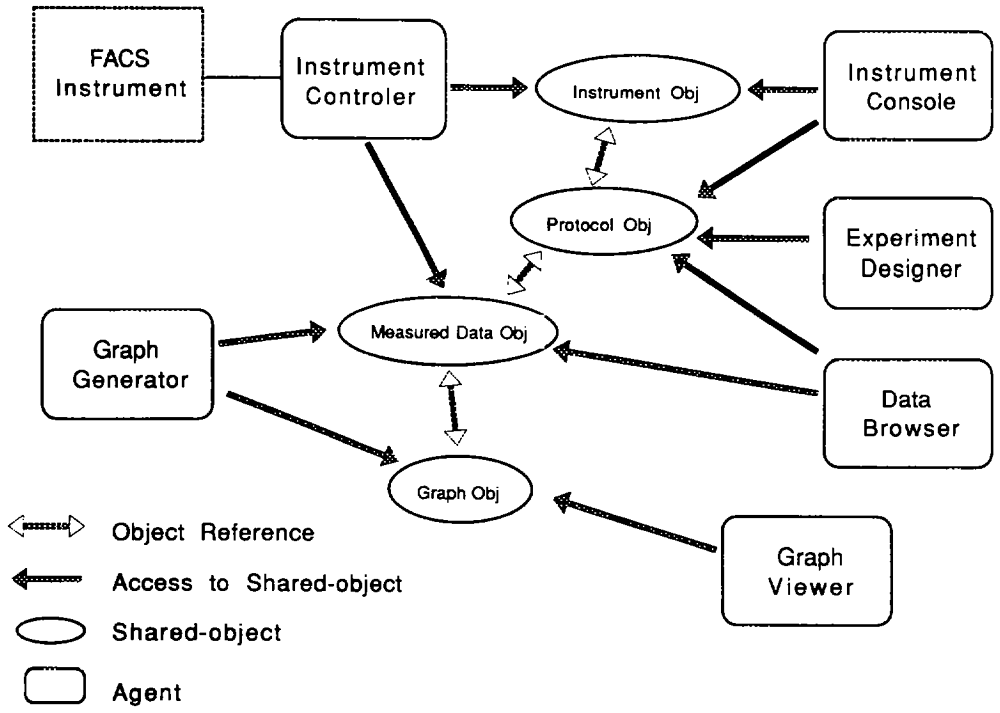

Figure 2. FACS Support System Structure

```
(defconcept reagent-seq
  (derived abstract) (isa sequence)
  ((type reagent)
     ;; meta attribute ''type" to support
     ;; parameterized class
     ;; This is clumsy and will be fixed.
   (elements literal-sequence))
  (restriction true))

(defconcept tube (base entity) (isa top)
  ((reagents reagent-seq) (cells cell-seq))
  (restriction
   (forall ((x (reagents self)) (y (reagents self)))
     (if (not-equal (antibody x) (antibody y))
         (not-equal (fluorochrome x)
                    (fluorochrome y)) ) ) ) )
```

The class **reagent** has attributes **antibody** and **fluorochrome**. When an attribute name of a class appears in an expression, it is interpreted as the accessor function. In the above example, the signature of **antibody** in the expression:

```
(not-equal (antibody x) (antibody y))
```

is "**reagent -> string**." The class **reagent-seq** is a kind of parameterized class that represents a sequence of reagents. The class **tube** represent a tube containing a mixture of cells and reagents. The constraint on the mixture is that *there should not be two reagents that have different antibody but have the same dye.* The quantifier variables can be associated with either class name of an entity-object class or **sequence**-valued expression. In the above example, **x** is associated with **reagent-seq**-valued **(reagents self)**; the type of **x** is **reagent**. If a quantifier variable is associated with a class name, it is interpreted to be associated with the sequence of all instances of the class.

Maintenance of Constraints Defined by First-Order Formulae When describing logical constraints among objects, we want to allow "arbitrary" first-order formulae to be attached to classes. However, if we do so, the maintenance of the logical constraints becomes computationally undecidable. We avoid this problem by the following scheme in which logical formulae are boolean functions. This is a practitioner's approach, but it works pretty well, especially when the number of objects is small(from a few hundred to a few thousand). As mentioned later, the operational semantics provide a more comprehensive programming interface than a rule-based approach.

1. Let all of the built-in predicates have negated counter parts. For example, the negation of **greaterThan(x,y)** is **lessOrEqualTo(x,y)**. All of the built-in predicates(boolean functions) have at most polynomial (time/space) complexity with respect to the size of the objectbase. (As a matter of fact, most of them have constant complexity.)
2. Assure the termination, when defining a new base predicate. In practice, we can implement almost all of the predicates with polynomial time complexity with respect to the size of objectbase. (**Quartz** allows a programmer to use logical operators and quantifiers in the definition of new predicates. However, the programmer must assure termination and reduce the complexity.)
3. Implement the negations of newly defined base predicates as other predicates (boolean functions). Although this step could be just a composition with not-function, it makes the optimization easier.
4. Attach first-order formulae to classes by constructing them out of the given base predicates using logical operators (including negation) and quantifiers. **Quartz**restricts quantified variables to iterate through only those objects in the database. Thus, the complexity of evaluating such formulae is bound by those of the base predicates; trivially, *if the base predicates have polynomial(time/space) complexity, so will the derived first-order formulae; if the evaluation of base predicates terminates, so will the evaluation of derived first-order formulae.*
5. Upon an update of the objectbase, do the update first, then evaluate the relevant formulae to check the consistency. If inconsistency is detected, roll back the update. The compiler determines which formulae should be checked upon which updates. In this way, we don't have to "infer" whether an update will induce inconsistency. (If we use a resolution-based inference procedure, the constraint maintenance is also undecidable.) In the current implementation of **Quartz**, the checking procedure is activated when a transaction is committed. Also, in order to allow finer granularity of checking, **Quartz** supports nested transactions.

This method encapsulates the execution semantics inside the implementation of base predicates. Thus, so long as only base predicates are used to construct first-order formulae, the description of the logical constraints is *completely* declarative as opposed to the situation in rule-based systems, where we are always haunted by the non-declarative execution semantics of inference engines, as discussed in the third section.

Of course, our method does have a drawback: without extensive optimization by the compiler, the complexity escalates when the quantifiers are deeply nested. The basic strategy of the optimization is essentially the same as that of relational query optimization: *reduce the numbers of objects that quantified variables go through by evaluating subexpressions in an appropriate order.*

Our distributed object environment introduces one more twist in the optimization of consistency maintenance. Since the attribute mappings (associative sets implemented by Btrees) are distributed over multiple machines, we have to ship out the subexpression evaluations in order to avoid loading the entire associative set onto the machine on which the evaluation request is issued. This technique is essentially the same as the query optimization over vertically-fragmented relational tables in a distributed relational database system [Ceri & Pelagatti 1984].

Communication via Shared Objects

First, we describe the initial implementation of the communication facility through shared-objects, which exploits an commercial object-oriented database (Objectivity[5]). Then, we discuss problems we encountered and how we are trying to fix them.

- Shared-Objects
  Shared-objects are implemented as objects of Objectivity. The distribution of objects is supported by Objectivity.
- Agent Management
  An agent is managed by its *service name*, which identifies the agent's function. When an agent wants to let another agent bind to a shared-object, it simply requests the other agent by its service name. For example, given an shared-object **facs_machine_obj** that represents a FACS instrument, a instrument controller agent would request that a graphical console panel be associated with the shared object as follows:

  ```
  facs_machine_obj.requestService("CONSOLE_PANEL");
  ```

  (**requestService** is a method of the base class of all shared-object classes.) Then, an agent (process) that displays the "console panel" of the machine would be created (if necessary), and associated with the machine object. After this, whenever the actual machine changes its status, the instrument controller agent will modify the machine object. The modification activates a triggered method in the "console panel agent" and updates the graphics on a screen, e.g. the "meters of the machine."
- Notifications and Triggered Methods
  The notification mechanism, which is the basis of triggered methods, is implemented with internet sockets. Basically, each agent is always listening to an internet socket for incoming notifications. The structure of a notification is just the object-identifier of the shared-object and the notification-ID.

  A notification is dispatched with respect to the class-ID of the shared-object and the notification-ID; the corresponding triggered procedure is activated upon the shared-object.

  When an agent issues notifications inside a transaction boundary, the system accumulates them and sends out the packets of notification to the perspective agents at the transaction commit time.

[5]Objectivity is a commercial object-oriented database system that has tight integration with C++ for object access and manipulation.

Although this initial implementation worked correctly, it turned out that the overhead of transaction management by the object-oriented database was too large. When an agent issued a notification (trigger), it took at least 400 milliseconds for the corresponding procedures to be activated in other agents(DECStation 5000/240, 56MB memory, local object server, local disk). Also, if two agents share an object and both are inside their transaction boundaries, neither agent can see an update performed the other agent, nor can the agents communicate by means of triggers., Since update of an object can occur only within a transaction boundary, this situation is inevitable so long as a full-fledged object-oriented database is used. Yet, we need to use the advantages of the object-oriented database, such as network transparency, machine architecture (byte-order) independence, and consistent interface with programming language (C++). Thus we decided to introduce a minimum mechanism to support shared-objects while still using the object-oriented database. More specifically, we use RPC to update directly on-memory images of shared-objects without going through Objectivity's update mechanism. Most of the programming interface for shared-object stays the same, except that notification *can* be sent at any time rather than at a commit time. A preliminary implementation showed that this scheme could attain a latency period as small as 2 milliseconds.

Abstract Instruments

As described earlier, we have three FACS instruments of three different hardware designs in our laboratory. We need to establish a method to control them in a homogeneous way.

In our system architecture, this is straightforward, because an instrument is visible as an instrument object (shared object) to all FACS agents besides the instrument controller agent (which must be hardware-specific). Thus, we can represent generic FACS instruments as an abstract class, **facs_instrument**, and define polymorphic triggering methods[6]. Each individual model of FACS instrument is represented by a class derived from the abstract class. This framework has some advantages:

- Uniform Interface to Instruments
  As intended, a FACS agent can access different FACS instruments in the same way, so long as it uses the polymorphic methods of **facs_instrument**. (Of course, the instrument controller agent should implement the instrument-specific triggered methods.) In this way, all the FACS instruments can appear as the same virtual instruments [Santori, M. 1990].
- Easy Extension of Instruments
  If we do need to access features unique to each individual instrument, we just define the methods deal-

[6]In C++, they are triggers implemented as virtual member functions.

ing with those features in the derived class. For a user, this may appear just as an additional portion of the graphical control panel. Thus we can preserve the consistency of the instrument operations as much as possible.

Discussion

Benefits for Biologists and Biological Laboratories

This subsection describes some of the benefits that our system architecture can provide to biologists or biological laboratories.

- Integration of Data from Different Sources
  Since all the information (data) are represented as objects in a uniform way, it is easy to build tools (agents) that process data objects from different sources. For example, the graph generator/viewer agent can deal with data objects generated from an agent controlling an instrument other than a FACS. Moreover, the agent architecture makes it very easy to extend the system; it is even possible to extend the system at run time by just starting new agents.

  From the biologist's perspective, when a new instrument is incorporated into the system, all the existing tools will be readily available to analyze the data generated from the instrument; when a new analysis tool is installed in the system, the tool can analyze the data generated by existing instruments, as long as the new agent supports the existing shared object structures.

- Centralized Monitoring of Instruments
  Fully exploiting the distributed abstract instrument objects, we can easily construct a centralized monitoring system for multiple instruments. More specifically, a console agent for each instrument runs on both the local controlling computer and the central monitoring computer (workstation); both console agents share the same instrument object. Since the central monitoring computer can run console agents for several instrument objects, an experienced operator can monitor the use of instruments while inexperienced users work with individual instruments.

  This framework is quite useful for a laboratory that has sophisticated instruments. For example, our laboratory has two full-time operators for our three FACS instruments (which are located in different rooms), because many biologists need operator attendance for operating the instruments. However, as nothing unusual happens most of the time, the operators often sit idly beside the instruments with the users, looking at the console and occasionally adjusting the parameters. Theoretically, one operator could easily handle three machines at the same time, if that operator had a comprehensive, centralized monitoring system.

  The framework also provides remote monitoring capability, which is important when hazardous materials are being used.

Comparison with Other Laboratory Automation Systems

In order to compare with an exhaustive list of references in the biomedical applications in recent years, the author searched the BIOSIS[7] database with the keyword `laboratory automation` in the title and abstract, resulting in 107 articles from the last five years' publications.

It turned out that most of these articles focus on special computational methods to support specific experiments. Only few of them address a general framework for constructing a generic laboratory infrastructure. Some of them address a networked environment but only discuss connectivity between PCs and mainframe/mini computers; even the conventional client-server computing model is not fully exploited. As far as the author could find out, none of these articles address a true distributed computing environment in which all the processes (agents) have "equal-rights" of their own; nor does any article discuss object-oriented data management in a distributed environment.

Thus, the author proceeded to search the INSPEC[8] database with the same keyword, resulting in 490 citations from the last five years. Of these references, 24 citations refer to either **`distributed system`** or `object-oriented` in their abstracts; 5 papers were of some relevance.

[Nelson & Alameda 1988] and [Cardoso, F.J.A. 1989] address networked instrument control and some basic communication architecture; however, a consistent distributed data modeling is missing.

[Zhou 1992] presents an expert system that supports the design of laboratory automation systems, which has good system modeling capabilities.

[Kachhwaha 1988] is about an object-oriented database with a relational database back-end, which supports laboratory automation. However, clear modeling of distributed processes and logical constraints is not provided.

[Couturier, J. F. 1992] presents a model of the quality control in manufacturing process using the E-R model. It addresses the *virtual manufacturing machines* connected over the network, which are essentially the same as our abstract instruments. However, the object-orientation and logical constraints representation is limited within the scope of the E-R model.

Thus, our system is unique in that it integrates distributed data management, distributed processes, communication among processes, and logical constraints maintenance into a consistent object-oriented

[7]BIOSIS is a reference database of publications in biomedical field.

[8]INSPEC collects scientific and engineering publications.

framework, using advanced techniques such as inter-agent communication via shared objects, and logical-constraint annotation of classes.

Conclusion

We have demonstrated that a distributed object system with an integrated constraint-maintenance mechanism and communication by shared-objects can provide a powerful infrastructure for biological applications. By attaching logical constraints to objects, the maintenance of constraints becomes simpler, because such an arrangement provides a natural way of localizing constraint semantics to the objects themselves. Furthermore, the concept of agents communicating via shared-objects serves as a comprehensive programming interface by encapsulating the details of network programming. The shared-object metaphor also simplifies the design of the communication protocol.

Nevertheless, our system is still in a preliminary stage, which requires further development such as:

- Optimization of the Triggering Mechanism
  In the current preliminary implementation, the granularity of cache coherence for shared objects is determined object-by-object. When a shared object has a large size, e.g several dozen kilobytes, it will degrade the performance. We need to build a compiler that will generate cache coherence maintenance procedures with finer granularity.
- Integration of **Quartz** with C++-based Objectivity
  As the syntax might have suggested, **Quartz** is a Lisp-based system, although it has its own networked object server written in C. Although **Quartz** can store objects in Objectivity as well, it is not well-integrated with the C++-based Objectivity environment. We should completely rewrite **Quartz** to be an extension of C++ class definitions. More specifically, we should build a preprocessor that generates Objectivity schema, RPC stubs for triggered methods, and procedures for logical constraints maintenance.
- Consistency Maintenance Over Replicated Objects
  We are exploring the possibility of making our computing facility available to biologists nationwide, letting them login to our system or letting their systems connect to ours. We need to replicate critical objects in order to reduce network overhead. In this context, the constraint maintenance algorithm should be completely rewritten considering replication and the different cost model.

Acknowledgment

The author would like to thank Joseph Norman and Prof. Larry Fagan for carefully reviewing this paper; Dr. David Parks, Marty Bigos and Wayne Moore for providing biological domain knowledge; Yuan GE and Neerav S. Parekh for programming the GUI of the system; Dick Stovel for providing the figure of sorting mechanism. Also, the author extends thanks to Prof. Gio Wiederhold for his encouragement; and to Prof. Leonore Herzenberg and Prof. Leonard Herzenberg for giving him the opportunity to work for them.

References

Barsalou, T. 1990. View objects for relational databases, Technical Report STAN-CS-90-1310, Dept. of Computer Science, Stanford Univ.

Barsalou, T.; Keller, A.M.; Siambela, N.; Wiederhold, G. 1991. Updating relational databases through object-based views, *SIGMOD Record*, 20(2): 248-257.

Cardoso, F.J.A. 1989. A distributed system for laboratory process automation, *IEEE transactions on Nuclear Science*, 36(5).

Ceri, S. and Pelagatti, G. 1984. Distributed databases : principles and system, McGraw-Hill.

Couturier, J. F. 1992. Information system and distributed control for a control cell architecture, In Proceedings of 1992 IEEE International Conference of Robotics and Automation, Los Alamitos, CA: IEEE Computer Society Press.

Kachhwaha, P.; Hogan, R. 1988. LCE: an object-oriented database application development tool. *SIGMOD Record*, 17(3): 207.

Matsushima, T. and Wiederhold, G. 1990. A Model of Object-identities and Values, Technical Report, STAN-CS-90-1304, Dept. of Computer Science, Stanford Univ.

Nelson, H.C; Alameda, G.K. 1998. Scheduled automation of laboratory displacement experiments from a computer workstation. In Proceedings of Petroleum Industry Applications of Microcomputers, 111-118, Rechardson, TX: Society of Petroleum Engineering.

Parks, D.R.; Herzenberg, L.A. 1989. Flow Cytometry and Fluorescence-Activated Cell Sorting, In Fundamental Immunology, Paul, W.E. ed., Raven Press.

Santori, M. 1990. An instrument that isn't really. *IEEE SPECTRUM* 27(8): 36-39.

Sujansky, W.; Zingmond, D.; Matsushima, T.; Barsalou, T. 1991. PENGUIN: an intelligent system for modeling and sharing declarative knowledge stored in relational databases, Technical Report, (IBMRD) RC 17367, International Business Machines Corporation (IBM).

Zhou, T.; Isenhour, T.L.; Zamfir-bleyberg, M.; Marshall, J.C. 1992. Object-oriented programming applied to laboratory automation. An icon-based user interface for the Analytical Director. *Journal of Chemical Information and Computer Sciences* 32(1): 79-87.