LAH379

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
Email matu@fermi.Stanford.EDU, Phone (415) 725-8437, Fax (415) 725-8564

February 14, 1993

Abstract

This paper describes a fully distributed biological-object system supporting FACS(Fluorescence Activated Cell Sorter) instrumentation. All component processes(instrument controller, protocol designer, data analyzer, data visualizer, etc) running on different machines are realized as "agents" working cooperatively. The communication among agents performed via shared-objects by activating triggered methods. This shared-object metaphor encapsulates the details of network programming. The system enables to annotate classes with first-order formulae that express logical constraints of objects, which are automatically maintained upon updates.

Key Words: Logical Constraint Maintenace, Object-Oriented Database, Distributed System, Shared Object.

Contents

1 Introduction

This section explains our application domain.

1.1 FACS Experiments

The FACS(fluorescence-activated cell sorter) is a basic device widely used in immunology and molecular biology to identify the specificity of protein expression on cell surface. In order to clarify the role of our FACS support system, we describe how FACS experiment is performed.

- Preparation

  1. Given cells of interest, we choose a set of reagents (known antibodies attached with specific fluorescence dyes), and mix them in a solution. If a specific protein pattern of cell surface matches the antibody, the reagents will stick to the cell. Choosing reagents is a complicated process that needs biological expertise.
  2. Adjust the configuration of FACS instrument to match the reagents. For example, the scaling factors of parameter measurement will be adjusted to particular reagents.

- Analyzing Cells

  1. The stained cells are conveyed in a microscopically thin fluid flow, and exposed to laser beams of specific wavelengths. By measuring light emitted from each cell when it passes the laser beam, we can determine how much of each reagent is bound to the cell, in cell-by-cell manner. 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 the computer system in many ways to identify cell subpopulations of biological significance. The subpopulations are represented by subdomains of the value space.

- Sorting Cells
  A specific cell subpopulation having been identified, FACS sorts out the cells that fall in the subpopulation.

  1. The subdomain information is loaded to the FACS instrument control system.
  2. same as the first step of analyzing cells.

3. Based on the measurement, the system determines whether the cell is in the 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 is controlled by the system. With elaborated synchronization mechanism, 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 different tube (**Figure 1. Sorting Mechanism**).
5. At the same time, measured values are stored in the computer system for further analysis and verification of the sorting process.

1.2 Issues and Requirements of FACS System

1.2.1 FACS Experiment Design

A design of a FACS experiment requires choosing a set of reagents for a given cell population and given property of those cells to observe, which is a highly complicated process. For example, the designer must consider

- what kind of reagents can identify which property of the cell
- what kinds of lasers installed in the machine, which fluorescent dyes are compatible with each laser wavelength; which band-pass filters are used.
- 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 given laser configuration. (For example, it is not useful to mix reagents of the different antibody with the same dye.)

FACS support system should store such knowledge in a knowledgebase, which includes logical rules that verify the correctness of reagent use. The experiment design tool should be able to access the current configuration(laser, filter, etc) of the machine.

1.2.2 Instrument Control

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

1.2.3 Distributed Data Management and Process Control

Currently, we are using several FACS instruments in different locations. The experiment specification, data produced by the experiments, even the instruments themselves should be represented and managed in a distributed environment. Also the processes should be distributed in order to utilize computer resources effectively. For example, computation intensive processes(data analysis, calculation for data visualization) should be done by high-end servers, while actual data display will be done by end-user workstations.

These 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*.

2 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.

2.1 Distributed Object System

As described in the previous section, the FACS domain knowledge(data) should be shared by multiple processes running on different machines. Thus, we should provide a distributed data management system. Also, in order to provide consistent and secure data management upon simultaneous updates, network failure, and process failure, we 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.

2.1.1 Relational Database v.s. Object-Oriented Database

FACS domain objects have complex structure 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).

When storing 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([BT 90], [BT 90]) until a couple of years ago. Although the idea of establishing a generic way of converting objects to/from relational database was interesting([BT 91]), it turned out that the overhead of the conversion was too large for our application [SW 91]. (In some cases, it took 20 minutes to construct objects out of tables with a couple thousand tuples, when outer-join was required. This may be resolved if commercial relational database vendors support outer-join. Yet, it is unlikely that we can get performance enough to be used in interactive use. The use of Penguin system is well-suited for the applications that load all data objects at the beginning of a user session and 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 on-memory images. The references among objects are efficiently converted by the database system with swizzling operation. Furthermore, the consistency of references is maintained by the database system.

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

2.1.2 Agents Communicating via Shared Objects

As discussed in Section 1.2, FACS agents communicate extensively to perform cooperative work. Thus, providing an abstract and comprehensive programming interface is critical.

Since we adopted 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, the all the agents that are "interested" in the object will be notified and the specific methods associated with the trigger will be 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 just act on the shared objects.

Moreover, this framework provides a better programming interface than traditional communication metaphors, such as message passing or RPC(Remote Procedure Call).

2.1.3 Message Passing/RPC v.s. Shared Objects

With a message passing scheme, such as using internet socket, a programmer has to explicitly parse objects out of stream or expand them into stream. With RPC, we need not parse objects. However, in both cases, if an agent wants to know the status of other agents, it must be explicitly included the communication protocol.[1]

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 that is hidden from programmers, 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 is getting larger in recent years.

One more advantage of the shared-object scheme is that we can use "conferencing metaphor" in designing protocol, rather than peer-to-peer communication or general broadcasting. This also makes the communication protocol simpler.

2.2 Built-in Constraints Maintenance

Since we have adopted the object-oriented framework, it is more consistent to facilitate the constraint maintenance mechanism for each object, rather than to have separate constraint maintenance module. Furthermore, describing constraints at the object level has the following advantages.

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

3 Technical Features of FACS Support System

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

3.1 System Structure

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

- Instrument Control Agent
  Controls a FACS instrument through direct connection; each FACS instrument has a dedicated control agent.

[1] When the author was working for Penguin system, he designed and implemented a message handler to simulate multi-threaded programming on VAX/VMS, so that each module of Penguin system can act as "agent." Although it worked fine, it was inconvenient that we had to design the communication protocol carefully, so that process status of agents can be figured out.

- Instrument Console Agent
  GUI(Graphic User Interface) for the Instrument Control Agent. It shares a *instrument object* with Instrument Control Agent.
- 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)
  Analyze data objects and generates *graph objects.*
- Data Viewer Agent
  GUI for displaying graph objects, which 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.)

3.2 Logical Constraint Maintenance

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

3.2.1 An Object-Oriented Data Description Language Quartz

Quartz is a simple object-oriented data description language with a CLOS[2]-like type system. Moreover, it allows the attachment of a first-order formula to each class in order to express constraints. **Quartz** compiles first-order formulae into procedures, which maintain the constraints automatically based on operational semantics. 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 of entity-objects are persistent, although some transient objects could reside only on memory for their entire life-time. Complex objects are represented as a collection 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 for **Quartz** and the details are described in [MA 90].

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

```
(defconcept reagent (base entity) (isa top)
  ((name string) (antibody string) (fluochrome string))
    ;; ((<attribute name> <attribute value type>) ... )
  (restriction true) )
    ;; (restriction <constraints formula>)

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

[2]Common Lisp Object System

```
(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 (fluochrome x) (fluochrome y)) ) ) ) )
```

The class `reagent` has attribute `name`, `antibody`, `flourchorme`. When an attribute name of a class appears in an expression, it is interpreted as the accessor function. For example, the signature of `antibody` in an expression 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 constrains of 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 or `sequence`-valued expression. In the above example, `x` is associated with `reagent-seq`-valued `(regents 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.

3.2.2 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 operational semantics. This is a practitioner's approach, but works pretty well, especially when the number of objects is small; this is the case in our application domain(only a few hundred objects are involved in evaluating a particular first-order formula). 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 (a boolean function). 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 arbitrary first-order formulae to classes by constructing them out of the given base predicates using logical operators(including negation) and quantifiers. 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 rule-based systems, in which we are always haunted by the non-declarative execution semantics of inference engines.

Of course, our method does have a drawback: without extensive optimization by the compiler, the complexity escalates when the quantifiers are nested in many levels. 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 that sent the evaluation request. This technique is essentially the same as the query optimization over vertically fragmented relational tables in a distributed relational database system [CS 84].

3.3 Communication via Shared Objects

First, we describe the initial implementation of the communication facility though shared-objects, by fully exploiting the object-oriented system(Objectivity). Then, we discuss problems we encountered and how we are trying to fix them.

- Shared-Objects
  The shared-objects are implemented as objects of Objectivity[3] The distribution of objects is supported by Objectivity.

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

- Agent Management
  An agent is managed by its *service name*, which identifies the agent's function. When a 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 machine control agent would request a graphical console panel associated with it:

  ```
  facs_machine_obj->requestService("CONSOLE_PANEL");
  ```

  (`requestService` is a method of base class of all shared-object class.) Then, an agent(process) that visualizes "console panel" of the machine will created (if necessary), and associated to the object. After this, whenever the actual machine changes its status, the machine control agent will modify the machine object. The modification activates a triggered method in the "console panel agent" and update the graphics on a screen, e.g. the "meter of the machine."

- Notifications and Triggered Methods
  The notification mechanism, which is the basis of triggered methods, is implemented using internet sockets. Basically, each agent is 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 it and sends out the packets of notification to the perspective agents at the transaction commit time.

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. Also, if two agents are sharing an object and both are inside their transaction boundaries, each cannot see an update performed the other agent, nor can they 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, consistent interface with programming language(C++), etc. 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 committed time. A preliminary implementation showed that this scheme could have a latency as small as 2 milliseconds.

4 Conclusion

We have demonstrated that a distributed object system with an integrated constraint maintenance mechanism and a communication metaphor 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. Further, the concept of agents communicating with 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.

Our system is still in a preliminary stage, and requires further development such as:

- Integrating **Quartz** with C++-based Objectivity
  As the syntax might have suggested, **Quartz** is lisp-based system, although it has its own networked object server written in C. Currently, **Quartz** can store objects in Objectivity. However, it is not well-integrated with the C++-based Objectivity environment. We will completely rewrite **Quartz** to be an an extension of C++ class definitions. More specifically, we will build a preprocessor that generates Objectivity schema and RPC stubs for triggered methods.
- Consistency Maintenance Over Replicated Objects
  We are exploring possibility of making our computing facility available to nationwide biologists, letting them login to our system or letting their system 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.

5 Acknowledgment

The author would like to thank Mr. Joe Norman and Prof. Larry Fagan for their carefully reviewing this paper. Also he thanks to Prof. Gio Wiederhold for his encouragement and support, Prof. Lee Herzenberg for giving him the oppotunity to work for her. This reseach was supported partly by NLM(National Library of Medicine) grant 2 R01-LM04336-04A1; the author expresses his gratitude for the support.

References

[BT 90] Barsalou, T.; Sujansky, W.; Herezenberg, L.; Wiederhold, G. "*Management of complex immunogenetics information using an enhanced relational model*", Stanford Technical Report, STAN-KSL-90-42, January 1992.

[BT 90] Barsalou, T. "*View objects for relational databases*", Stanford Technical Report STAN-CS-90-1310, March 1990.

[BT 91] Barsalou, T.; Keller, A.M.; Siambela, N.; Wiederhold, G. "*Updating relational databases through object-based views*, Proceeding: 1991 ACM SIGMOD International Conference on Management of Data, May, 1991.

[MA 90] Matsushima, T. and Wiederhold, G. "*A Model of Object-identities and Values*" Stanford Technical Report, STAN-CS-90-1304, February 1990.

[PD 89] Parks, D.R.; Herzenberg, L.A., "*Flow Cytometry and Fluorescence-Activated Cell Sorting*", in "*Fundamental Immunology*", ed. Paul, W.E., Raven Press Ltd. 1989.

[SW 91] Sujansky, W.; Zingmond, D.; Matsushima, T.; Barsalou, T. "*PENGUIN: an intelligent system for modeling and sharing declarative knowledge stored in relational databases* " International Business Machines Corporation (IBM). Technical Report, (IBMRD) RC 17367, 1991.

[CS 84] Ceri, S. and Pelagatti, G. "*Distributed databases : principles and system*, McGraw-Hill, 1984.