AN ACTIVE OODB SYSTEM FOR GENOME PHYSICAL MAP ASSEMBLY[*]

A. J. Lee[†], E. A. Rundensteiner[†], and S. Thomas[‡]

(†) Software Systems Research Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122
e-mail: rundenst@eecs.umich.edu
telephone: (313) 936-2971
fax: (313) 763-1503

(‡) Information Technology and Networking
University of Michigan Medical Center
Human Genome Center, 2570C MSRB II
e-mail: spencer@eecs.umich.edu
telephone: (313) 764-8065
fax: (313) 764-4133

Abstract

In this paper, we describe the design and implementation of a scientific database for the map assembly tasks performed by the geneticists at the University of Michigan Human Genome Center. Our system not only manages complex genomic data, but it also supports the automation of the associated map assembly tasks. For the former, we present a genomic object model that integrates both experimental and derived data. For the latter, we describe operators to automate some of the analysis steps, such as inferring overlap information using transitivity rules. Strongly motivated by the need of the physical map assembly task for both inferencing capabilities as well as object modeling power, we have designed and implemented an active object-oriented database (OODB) system, called Crystal, on the GemStone OODB. Crystal seamlessly integrates rule inferencing with complex object modeling and other typical database capabilities, thus avoiding the overhead in moving data between systems for rule processing and data management, and eliminating data mismatch in these different representations. Crystal executes inferencing automatically, without explicitly having to initiate rule execution, and responds to the incremental addition of new experimental data. In this paper, we also discuss the implementation of the genome information model and physical map assembly inferencing system on top of Crystal. In conclusion, we provide a walk-through example and several experimental results that demonstrate how our approach can be used to effectively support physical contig assembly.

[*]The authors are grateful for partial financial support from the NIH under grant NIH #P30HG00209-03.

1 INTRODUCTION

The Human Genome Project is an international 15-year effort to map all the genes in the human genome and to use that information for diagnosis and treatment of genetic diseases such as cystic fibrosis. There are roughly 100,000 genes in each human cell that make up the human genome. Genes encode information about how and when to form the proteins that govern the daily functioning of the human body, and range in size from about 1,000 to 2 million base pairs long. The physical mapping process [4] first separates the chromosomes from the nucleus, then breaks them up into smaller pieces, which are inserted into the genome of lower organisms, e.g., bacteria, where they can be replicated rapidly and in large amounts. The resulting DNA fragments (clones) are then used for future experiments. The purpose of the experiments is to identify features of the DNA fragments and determine the locations of these features on a chromosome by ordering the fragments to their respective locations on the chromosomes. In short, the scope and complexity of the Human Genome Project is such that previously acceptable manual methods are no longer suitable. It thus is vital that we develop software systems that aid genomic experimental efforts by managing the volume of complex genetic data sets and related meta data and by automating labor-intensive analysis tasks, whenever possible. Database design is one important component of this effort, with the goals of freeing geneticists from the tedious tasks of data organization, search and retrieval, and of giving them access to the human genome data and associated analysis tools in a user-friendly and efficient fashion.

In this paper, we describe the design and implementation of a scientific database system for supporting this physical map assembly task performed by scientists of the Michigan Human Genome Center. We first introduce the genome object model [18] we designed to capture the genomic objects and their interrelationships. These range from raw experimental data over analytically derived data, up to metadata describing, for instance, the data sources. Our model is developed using object-oriented database (OODB) technology [5] since the latter provides powerful modeling constructs for capturing such complex genomic information naturally and efficiently. We are interested in supporting the task of creating a *contig map*, assembling overlapping DNA fragments into a contiguous stretch. The concept of ordering and orientation is thus fundamental to our model. Our genome object schema efficiently supports the maintenance of unordered, partially ordered, and completely ordered sets of data based on an overlap refinement hierarchy. The relative ordering information we get from experiments may not be orientable with respect to the global ordering viewed at the chromosome level. Our model thus supports the concept of a local orientation frame that represents the relative orientation of an ordering with respect to the global view. Additionally, we describe sets of operators we have developed to automate analysis steps and map construction steps. Examples are operators for combining information in multiple frames into a single frame, and for refining ordering relationships to derive more precise relationships.

To reduce the time necessary for analyzing large amounts of genomic experimental data as well as to react efficiently to the insertion of new experimental data by scientists, we propose to solve the physical map assembly problem using a rule-based approach. While recently several approaches for developing support tools for physical map assembly have been proposed in the literature, they typically use heuristic or optimization approaches [21, 25]. The rule-based approach is appropriate for several reasons. First, the domain is data-driven, that is, we can use the available set of known facts (i.e., data about the experimental data) to draw further conclusions about fragment orderings. Second, the experimental data is often uncertain and ambiguous. A known advantage of rule-based systems is their ability to deal with uncertain data [23], i.e., it is straightforward to incorporate certainty factors into the rule-based system. Third, the mapping process is not yet completely understood. We thus have elicited typical actions done by scientists in the process of analyzing experimental results and deducing fragment positioning and ordering in a map, and we have explicitly formulated them as rules. By automating the physical mapping task, we will minimize or even eliminate the clerical and other human errors, and more importantly we will be able to free the time of biologists for the more difficult problems of genomic mappings.

We first developed a system composed of a separate expert system and database system to demonstrate the feasibility of our approach. However, this has led to several problems including the potential mismatch of data in both systems and the overhead in moving the data between the systems. To overcome these problems, we now have built an active system, Crystal, that integrates rules into an OODBMS. In Crystal inferencing can be done automatically, without explicitly having to initiate rule execution by the user. We have focused on the efficient evaluation of production style of rules, since these appear to be relevant to the task at hand. In this context, we have explored subscription and incremental rule evaluation strategies to optimize rule processing in large genomic databases. The first prototype of the active OODB system Crystal is successfully running in the University of Michigan Software Systems Research Laboratory.

We have implemented the genome physical mapping model and contig construction tool using Crystal, and we have run real genomic data sets through the physical map assembler tool. To the best of our knowledge, this is the first time that an active OODB system has been applied to solve genomic application tasks. While we have developed our active OODB system to solve the map assembly task, Crystal is a general-purpose design, and thus will also be applicable for other scientific applications. In this paper, we present through two comprehensive examples which have been successfully run on our system to demonstrate feasibility of our approach. We also report experimental results from applying our system to several genomic data sets.

The contributions of our work presented in this paper can be summarized as follows:

- Develop a genomic object model that supports the efficient representation of both precisely and partially known ordering data, integrates both experimental and derived data for support of the physical map assembly task, and allows incremental development of the database as new experimental data is added.
- Define an associated set of rules for fragment ordering and physical map construction, and provide for the identification of inconsistent information in ordering relationships and the potential of resolution of such situations.
- Design and implement a seamlessly integrated active OODB system, Crystal, that supports rule inferencing, complex object modeling, and typical database capabilities.
- Build the physical map assembly tool based on the genomic object model and associated physical map assembly inferencing tasks using Crystal, such that the derivation of the physical map can be directly supported within the context of our active database system.

The remainder of this paper is structured as follows. In Section 2, we introduce the genomic object model we have designed for the physical map assembly task, the associated set of inferencing rules we have developed to automate the inferencing steps, and the implementation requirements for building such a system. Section 3 describes the design and implementation of Crystal, a system that integrates rule capabilities with object technology. Section 4 discusses how to build a physical map assembler tool upon Crystal. Section 5 goes through two comprehensive examples and shows experimental results from applying our system to several genomic data sets. We discuss related work in Section 6, and present conclusions and future work in Section 7.

2 DESIGN OF THE PHYSICAL MAP ASSEMBLER

Physical mapping, especially contig assembly, refers to the process of assembling overlapping clones to cover a contiguous region of the genome. Genomic experiments reveal pairwise or higher-order relationships between genomic pieces of DNA fragments, such as loci, markers, genes, YACs, cosmids, STSs, etc [4]. In this task, genome researchers have to deal with massive amounts of complex genomic data from experiments and analysis. The concept of ordering and orientation is fundamental to our system. They need to address the problem of relative ordering: when a chromosome is broken into pieces, we cannot tell which way a DNA fragment is oriented with respect to the chromosome. Furthermore, the experiments are done on a collection of DNA fragments, and we thus can determine their ordering relationship only relative to this respective collection. In short, the relative ordering information we get from experiments may not be orientable with respect to the global ordering viewed at the chromosome level. Our model thus supports the concept of a *local orientation frame* that represents the relative orientation of ordering with respect to the global view.

Given an initial set of pairwise ordering relationships between DNA fragments (i.e., the raw experimental data), we are interested in deriving as many as possible ordering relationships. This kind of experiment is clearly very expensive and time consuming to perform, and we thus have developed database tools to automate the mundane aspects of map assembly, whenever possible. After we get the transitive closure of the initial data set, we are interested in creating a contig map, assembling overlapping DNA fragments into a contiguous stretch. The quantity of data, the complexity of the inter-relationships between genetic and genomic elements, and the fluidity of the concepts (e.g. a particular clone may, at various times, be a target of a probe, a probe, a fragment of DNA, point-like, a single element, or the "parent" of a library of smaller clones) demand a data modeling and representation technique more powerful than traditional databases. We thus develop a genomic data representation model utilizing object-oriented database (OODB) technology. Object-oriented data modeling provides a solid foundation for capturing the complexity of the genome application domain, and in addition OODBs also support the much needed capabilities of data retrieval and data manipulation.

2.1 The Physical Map Assembly Object Model

In this section we introduce the genomic object-oriented data model we have developed for modeling relevant genetic concepts. The object model (Figure 1)[1] can be roughly divided into two parts. The left hand side covers concepts related to the genomic experiments. The right hand side corresponds to the map ordering information, i.e., it covers concepts related to generating and maintaining physical maps.

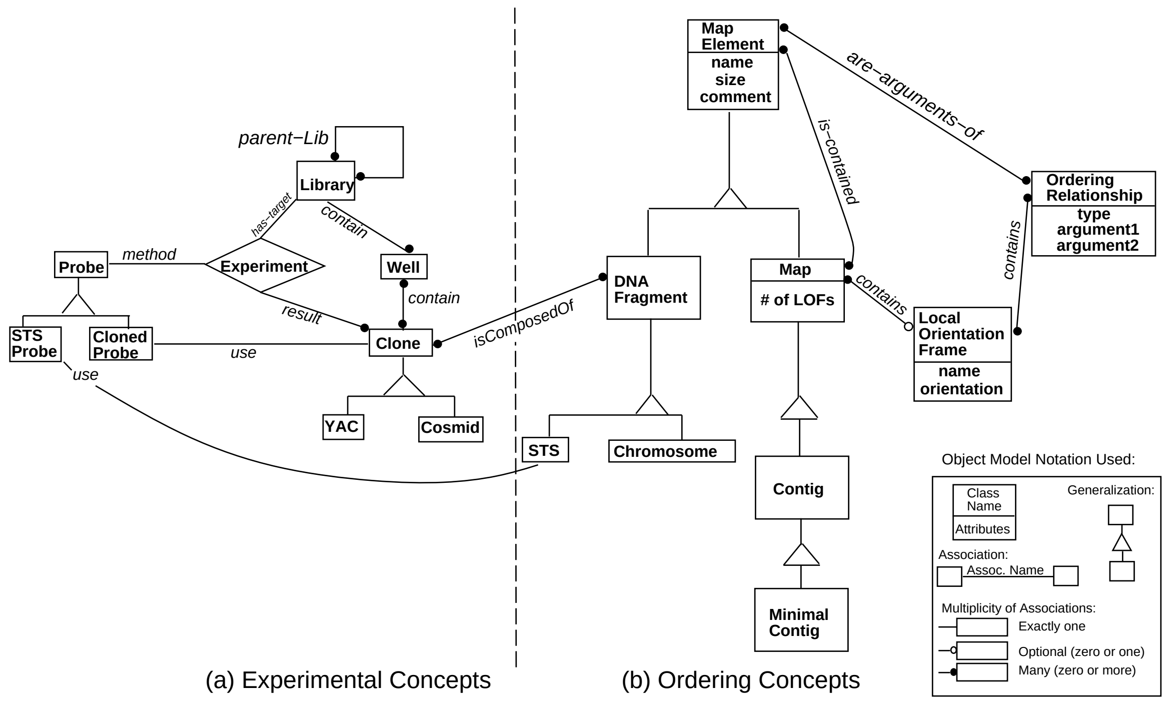

Figure 1: The Physical Map Assembly Object Model.

The object model necessary to capture genetic experiments consists of several classes, including Library, Probe, and Clone. A library is a collection of unordered clones, contained in wells. A well is a physical device that can hold zero or more clones. A clone is colony of organisms with identical genetic makeup, or the DNA fragment(s) of interest from such a colony. A clone may reside in one or more wells, because the cloning process is a random process. A clone can be further refined into classes such as YAC (Yeast Artificial Chromosome) and Cosmid. In order to model internal deletions in clones, a clone may be composed of more than one DNA fragment. The Experiment class is a ternary association among Probe, Library, and Clone classes. In any experiment, a probe is used against a target library, and the result is a set of clones that react positively to the probe.

The ordering side of the genome object model (Figure 1) contains the classes, Map Element, DNA Fragment, Ordering Relationship, Local Orientation Frame (LOF), Map, Contig, and Minimum Contig. DNA Fragment and Map are subclasses of Map Element. By creating Map as a subclass of Map Element, the system is able to construct a hierarchical map. That is, an element of a map can be a map itself. A map element may not have a physical instantiation, while a library element always has a physical instantiation and resides in some well. A Local Orientation Frame (LOF) is used to group a set of ordering relationships that are known to have the same orientation. There are three possible orientation values for a LOF: toward

[1]Some attributes and methods of the classes are omitted for clarity.

north, toward south, or unknown. When the orientation is *toward north*, it means clones are ordered in the direction from the q-arm to the p-arm of a chromosome; *toward south* means clones are ordered from the p-arm to the q-arm of a chromosome. A map contains a set of ordered/unordered map elements, and the ordering relationships between these map elements are grouped into LOFs. Although a library can be viewed as a degenerate map, we don't intend to find the ordering relationships among the library elements. On the other hand, when we say a map contains a set of map elements, we either know some ordering relationships among the map elements or we intend to find the ordering relationships among the map elements in the future. A DNA fragment is a contiguous part of a chromosome, while a clone is a contiguous sequence of base pairs. A clone could be composed of several DNA fragments that form an ordered set (which happens when a clone has internal deletion). Until we prove otherwise, we assume a clone is a DNA fragment. A contig is a set of contiguous overlapping clones. It can be viewed as a map or a DNA fragment. A minimal contig uses a minimal number of DNA fragments to cover a contiguous region of a chromosome.

The binary ordering relationship between two DNA fragments is represented as an object (Figure 1). It may come from experiments, from external sources, or may be derived by the system. We have developed an ordering refinement hierarchy (Figure 2) which we utilize to classify ordering relationships for genomic intervals.

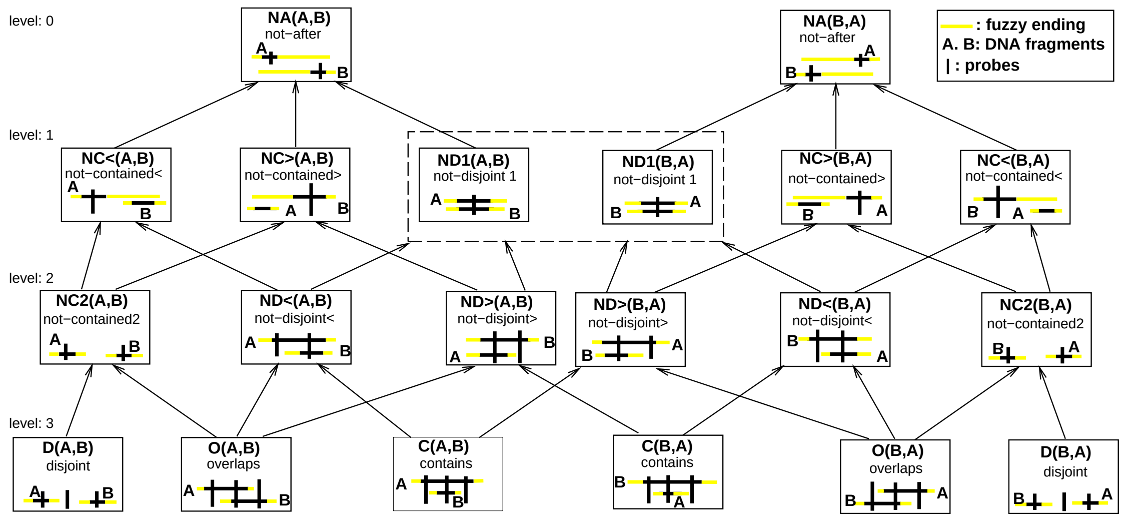

Figure 2: The Abstraction Hierarchy of Interval Relationships.

The refinement hierarchy of interval relationships [2] is based on our analysis of the type of experiments conducted with DNA fragments. For example, if we know that two intervals A and B share one common probe P1, then we know that A and B are not disjoint, denoted by *not-disjoint-1(A,B)* in our model. If we know that in addition there is one probe P2 that hits interval A but not interval B, then we know that A and B are not disjoint and that A cannot be contained in B, denoted by *not-disjoint<(A,B)*. The refinement hierarchy has several nice properties. (1) It supports both *precise* as well as *partial* information about interval relationships by using a single relationship only; and (2) the representation is *complete*, that is, the composition of any combination of these interval relationships will again result in one of the relationships contained in the abstraction hierarchy.

When we view DNA fragments and probes as intervals, the ordering refinement hierarchy we have designed is related to the work in the temporal reasoning domain [1]. Temporal reasoning deals with events and relationships between events. However, in the temporal domain there is a simple *global* orientation. This is different from the genome domain, where we are more likely to be able to infer only *relative* ordering information. The overlap and ordering relationships of DNA fragments are specified within some *local orientation* for the following reasons: first, a chromosome is broken into pieces and we thus cannot tell which side of a DNA fragment is left or right with respect to the chromosome, and second, the experiments

[2] We denote the start and end points of an interval A by start(A) and end(A), respectively. We use the following convention for expressing relationships between two intervals A and B: For each relationship rel(A,B), we have either (1) *start(A)* $<$ *start (B)*, (2) *end(A)* $<$ *end(B)*, or (3) *start(A)* $<$ *end(B)*. (1) has the highest priority, (2) the next highest, and (3) has the lowest priority.

are done on a collection of DNA fragments, so we will only determine ordering relative to this respective collection. Since this local orientation may not conform with the global orientation, we have introduced the local orientation frame concept (LOF). The *local orientation frame* is similar in spirit to the orientation windows introduced in CPROP [19] for orderings between points. A LOF allows us to represent the knowledge that a set of relationships exhibits the same orientation, without having to specify whether this orientation corresponds to the original orientation of the complete chromosome. An ordering relationship with respect to a LOF represents a relative ordering in the global view.

We express an ordering relationship between two DNA fragments as a pairwise constraint between fragments (as is typically done in constraint propagation systems). This allows us to map our genomic model to a simple graph representation. We construct a graph by mapping each DNA fragment to a node of the graph and each ordering relationship between fragments to an arc connecting the two respective nodes. For instance, the relationship (A before B before C) will be represented by the graph in Figure 3.a.

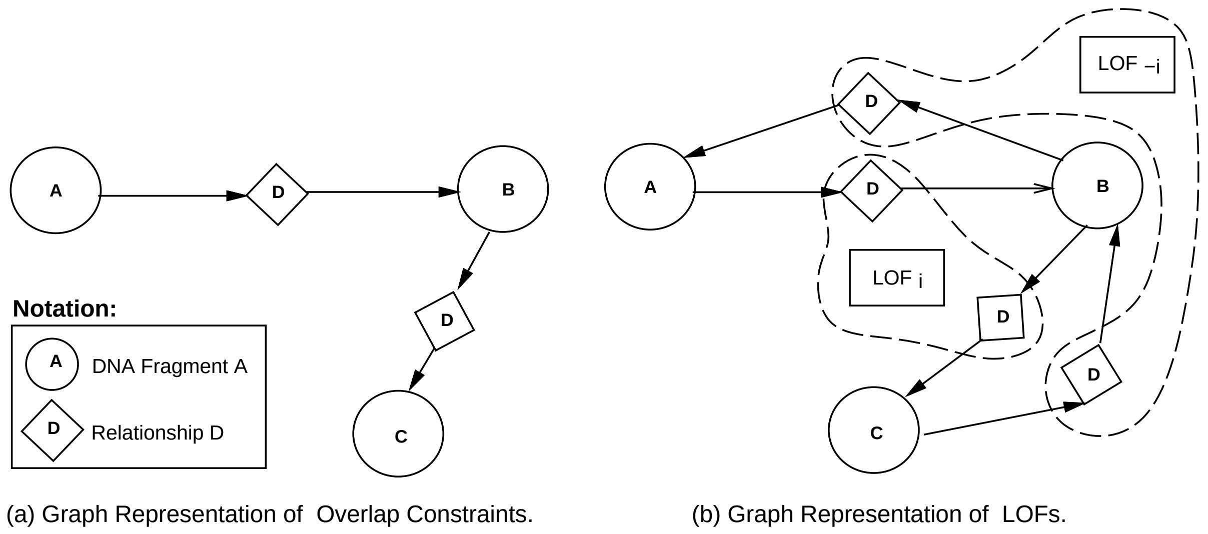
A
D
B
D
C
Notation:
A DNA Fragment A
D Relationship D
(a) Graph Representation of Overlap Constraints.
LOF −i
D
A
D
B
LOF i
D
D
C
(b) Graph Representation of LOFs.

Figure 3: Graph Representation of a Local Orientation Frame.

If we want to represent ((A before B before C) or (C before B before A)), then we need to be able to capture the associations among the four binary relationships in the graph structure. For this purpose, we introduce a grouping construct that combines sets of interval relationships that are known to have the same orientation (Figure 3.b). This grouping construct corresponds to the *local orientation frame* (LOF) introduced above. An ordering relationship is now said to belong to a *local orientation frame*. For instance, we would describe the situation in Figure 3.b by [*"A disjoint B" in LOF$_i$*] and [*"B disjoint C" in LOF$_i$*].

For a given *local orientation frame*, LOF$_i$, we may not know which of its two possible orientations is correct; the mirror image of a LOF$_i$ is denoted by LOF$_{-i}$. Including the mirror images into the system will allow us to apply inferencing rules without having to explicitly consider the inverse cases, thus simplifying our inference operators. The mirror image of LOF$_i$ is constructed by first building the mirror image of each individual interval relationship and then by combining them into a new frame, LOF$_{-i}$. For instance, for point relationships, if [*"A before B" in LOF$_i$*] then [*"B before A" in LOF$_{-i}$*]. For interval relationships, we need to determine the inverse of each of the relationships. The inverse of each relationship can be determined (1) by inverting the arguments of the relationship (e.g., from (A,B) to (B,A)) and (2) by replacing the relationship by its inverse (e.g., replace ND< by ND>).

2.2 Genomic Ordering and Orientation Inferencing

Next, we present mechanisms we have developed to support the inferencing of new ordering information based on experimental data stored in the genome database. Our goals are: (1) to keep the number of ordering relationships as small as possible in each LOF; (2) to reduce the number of LOFs in the database whenever it is possible; and (3) to infer as much as possible new information about fragment ordering. The total number of ordering relationships in the system and the complexity of the physical map assembly task have a positive correlation. If the total number of ordering relationships in the database is N, then the complexity of the task is O(N**2) on calculating the transitive closure only. Therefore, it is desirable to keep the number of ordering relationships in the system small. To achieve the first goal, we remove any less precise ordering relationships from a LOF, when a more precise ordering relationship is inserted into the LOF. Ideally all ordering relationships would be combined into one LOF with fixed orientation with respect to the chromosome, i.e., the *global orientation frame.* This would give us maximal new information about fragment ordering. To support this derivation process, we have developed three types of rules: relationship refinement rules, LOF composition rules, and transitivity rules. We are continuing to refine our system by extending this basic rule set, as necessary.

2.2.1 Relationship Refinement Rules

Relationship refinement rules are used to infer more precise ordering information about a pair of intervals in a single LOF. Each rule considers two relationships in the same LOF with the same pair of arguments. The arguments of these two relationships may or may not have the same order. For example in Figure 4, if we know *not-contained->(A,B)* and *not-contained-<(A,B)* in LOF_i, then we are able to infer *not-contained-2(A,B)* in LOF_i and retract *not-contained->(A,B)* and *not-contained-<(A,B)* from LOF_i. This rule is derived from Figure 2, since *not-contained-2(A,B)* is the highest common descendant of *not-contained->(A,B)* and *not-contained-<(A,B)*. The total number of the ordering relationships in the database is reduced each time after successfully firing a relationship refinement rule.

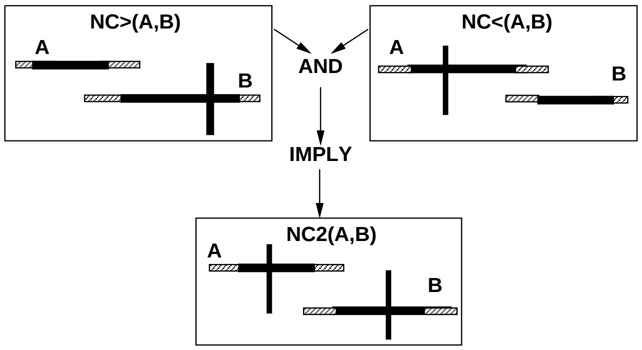

Figure 4: An Example of a Relationship Refinement Rule for Interval Relationships.

2.2.2 LOF Composition Rules

As stated above, it is our goal to derive as completely as possible the ordering relationships between fragments. Hence, we want to combine LOFs into more informative sets, whenever possible. The LOF composition rules consider two relationships in different LOFs with the same pair of arguments. To merge two LOFs containing ordering relationships between intervals, we need to identify one pair of intervals that belongs to both LOFs, say the pair (A,B) in Example 1 in Figure 5. Next, we need to determine whether the relationships of the pair (A,B) in LOF_i and in LOF_j together fix the same orientation. Since *not-disjoint-<(A,B)* and *not-disjoint->(A,B)* have a common descendant in the abstraction hierarchy (Figure 2), i.e., they together refine to *overlap(A,B)*, the two LOFs can now be combined into one LOF, denoted by $\mathrm{LOF}_{i,j}$. By inspection, it can be seen that the *overlap(A,B)* relationship must hold in the combined $\mathrm{LOF}_{i,j}$. A complete set of these rules can be found in [18] (see also Appendix).

Firing LOF composition rules always reduces the total number of ordering relationships in the database at least by one, because a LOF composition rule is based on two ordering relationships either having the same orientation or the opposite orientation. After LOF_i is combined with LOF_j, one of these (redundant) relationships can be retracted. In addition, we may be able to retract more ordering relationships from the newly formed LOF, if some ordering relationships become less precise than other existing ones in the new LOF.

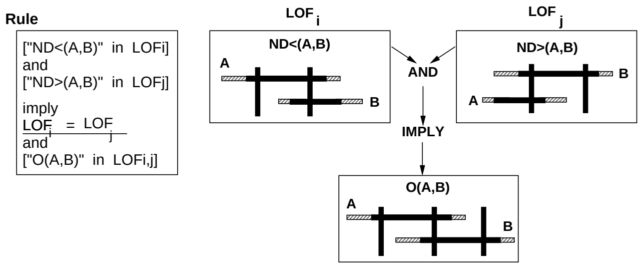

Figure 5: An Example of Combining Local Orientation Frames.

2.2.3 Transitivity Rules

Transitivity rules are used to infer additional ordering information about intervals in a single LOF. The transitivity rules consider two relationships in the same LOF, with at least one argument in common. These rules have the following type: *"(A rel1 B) and (B rel2 C) implies (A rel3 C)"*. One example, also shown in Figure 6, is:

IF rel1 is *overlap(A,B)* in LOF$_i$, and rel2 is *overlap(B,C)* in LOF$_i$
THEN rel3 = *not-contained-2(A,C)* can be added into LOF$_i$.

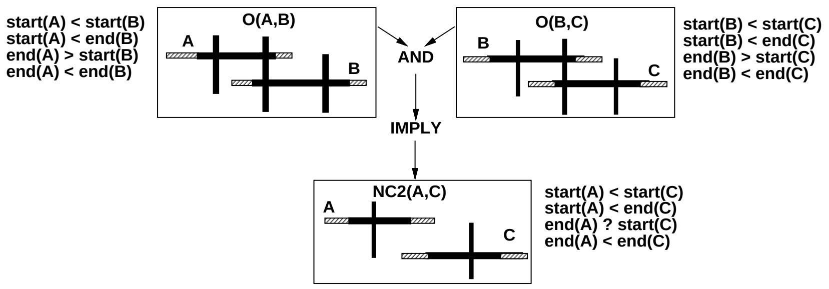

Figure 6: An Example of a Transitivity Rule for Interval Relationships.

Transitivity rules can be derived based on the known endpoint relationships between the respective intervals. This is also illustrated in Figure 6. In order to keep the total number of ordering relationships as low as possible, if there exists a more precise ordering relationship between the intervals A and C, then the new relationship *not-contained-2(A,C)* will not be added into LOF$_i$. On the other hand, if there exist only less precise ordering relationships between A and C, then the less precise ordering relationships will be retracted from LOF$_i$ when *not-contained-2(A,C)* is inserted into LOF$_i$. The total number of ordering relationships may not be reduced after firing a transitivity rule. At the worst case, the total number of ordering relationships in the system will increase by one each time. Firing transitivity rules alone may not be very interesting. However, the newly derived ordering relationships may in turn trigger other (relationship refinement, LOF composition, and transitivity) rules to be fired.

2.3 Contig Map Construction

Our system will derive the maximal information about the ordering relationships stored in the database, given the operators described in the previous section. While this forms the foundation of discovering further relationships, it would be too confusing to present the fully elaborated information to the user. For example, for a set of n completely ordered disjoint intervals, the resulting transitive closure would contain $n \times (n-1)/2$ disjoint relationships. It is sufficient, however, to give the $n - 1$ relationships between adjacent intervals. Thus, the redundant relationships should be pruned.

One way to simplify information is *minimal contig* construction. The goals of minimal contig construction are: 1) to cover the *largest possible contiguous regions* of the chromosome and 2) to use the *minimal number* of intervals needed to construct the map. We have made the following assumptions for contig map construction:

1. Maps are built by using known connectivity. That is, if we don't know whether two DNA fragments overlap or not, we make the pessimistic assumption that they do not. Therefore, contig maps are based on the *overlaps*, *not-disjoint-<*, and *not-disjoint->* relationships only. *Not-disjoint-1* and *contains* relationships cannot extend the map, and thus they are not used.
2. We assume all DNA fragments in a library will have similar lengths. Hence, for *not-disjoint->(A,B)*, we assume that the DNA fragment A is before the DNA fragment B in the contig, and vice versa, for *not-disjoint-<(A,B)*.

We have designed a simple algorithm for minimal contig construction that performs the following tasks in a local orientation frame (lof) to return contig(s) with more than one DNA fragment in each contig:

```
For each LOF in the database do:
1. Build a subgraph G (a directed acyclic graph) for the LOF consisting only
   of ND>, ND<, and O relationships and their associated intervals.
   The orientation of the edge corresponding to each relationship R(A,B)
   is from the left argument A to the right argument B.
2. While nodes in G are not marked:
   a. From a left-most interval (node), use breadth-first search to mark
      each connected node with its shortest distance from the starting node.
   b. Extract a minimal path between the starting node and a right-most
      connected node; this is a contig.
3. Order the contigs generated by step 2, if possible, using the disjoint
   relationships between their members.
```

An example of applying this algorithm to construct a minimal contig is described in Section 5.

2.4 Implementation Requirements for the Physical Map Assembler Tool

Based on the discussion in previous sections, we have identified powerful data modeling technology and inferencing functionality as the most important requirements for the physical map assembly task. We thus must construct a system to provide both capabilities.

If we were to use conventional database technology to handle the data modeling and query capabilities (Figure 7.a), then this would cause several problems as outlined below. Assume that the genome database stores pairwise ordering relationships between elements entered in our database, and we want to derive as many as possible ordering relationships from the initial data set. For example, in Figure 7.a, we have *($A < B$)* and *($B < C$)* in the data set, and we would like to infer *($A < C$)*. To infer new results using a traditional passive database, a user must either write her own inferencing programs or she must translate the data and ship it to a separate expert system. Application programs have to periodically query the database in order to check whether any changes in the data set warrant the re-execution of analysis tools. This clearly reduces adherence to the requirement of timeliness and may introduce extra overhead in translating data to and from the format used by an expert system.

In contract, if an active database system is used, the inferencing knowledge can be stored as part of the data set, and is automatically shared by the users (Figure 7.b). The system will infer new information immediately as new data or results are added. The user may then choose to keep the derived information or to commit the new data without saving the derived information.

Therefore, we propose to build an *active* OODB system integrating inference capabilities with OO data modeling power to support the physical map assembly task. In an active OODB, activities can be represented directly in a declarative format in the database by treating rules as first class objects. Given that the database is in control of both rule and data management, it can effectively optimize rule processing for all applications. This is an increasingly critical issue given the expected size of scientific databases, such as those maintaining the human genome data [13].

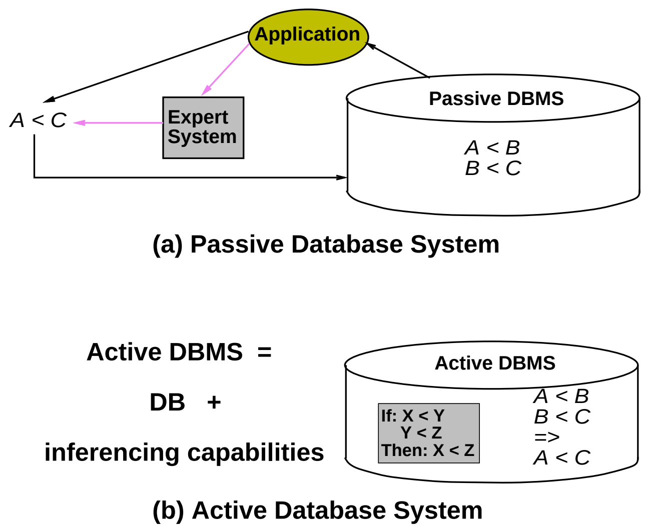

Figure 7: Passive Database System vs Active Database System.

3 CRYSTAL: THE ACTIVE GEM

Strongly motivated by the need of the physical map assembly task for both inference capabilities as well as object modeling support as outlined in the previous section, we now describe how we incorporate production rules into object-oriented technology. Unfortunately, commercial OODBs do not yet provide inferencing capabilities. We have thus designed and implemented an active database system customized to our scientific application. We have chosen GemStone[3] as implementation platform for this purpose, since it is one of the oldest and most mature OODB products and also since it has become an unofficial genome database standard - having been adopted by several other genome researchers. Our system Crystal[4], which supports a seamless integration of inference capabilities with object-oriented technology, is discussed below. While initially we have targeted Crystal for supporting the map assembly task, we plan to also exploit this technology as a principled approach for (1) enforcing arbitrary integrity constraints on genomic data, and for (2) monitoring external and internal events and notifying scientists if a situation of interest to them occurs.

3.1 Active OODB Model of Crystal

To meet the requirements of the physical map assembly task, i.e., to infer new information from the given data set, the application system has to monitor the state of the database as well as operations on the database. Because the classical Condition-Action (CA) rules are sufficient for handling the data-driven task such as computing the transitive closure from a data set, we focus on building an active OODB system, Crystal, that supports such CA rules. Other activity formats, such as triggers, alerters, and Event-Condition-Action (ECA) rules that are not required for the physical mapping task, are not discussed in this project, but could be easily added to our system in the future.

The fundamental design decisions required to build Crystal can be summarized as follows:

- **Rule integration with object modeling.** There are several alternative design choices for rules, namely rules can be (1) declared within class definitions, (2) specified as data member values, or (3) treated as independent objects [2]. The main disadvantage of the first and second alternative is that the rule's existence is dependent on the existence of other objects. Therefore, rules cannot be created, modified, and deleted at runtime. Furthermore, they could not continue to exist if changes on the class definition became necessary. For this reason, we choose the third alternative of treating rules as first

[3] GemStone is a registered trademark of Servio Corporation.

[4] The name Crystal of our active DBMS stands for *active gem*.

class objects. In Crystal, (1) rules can thus be created, modified, and deleted dynamically like any other object, and (2) furthermore a rule can monitor instances belonging to multiple classes. Note that the first two options cannot achieve this without losing modularity. For example, using the first approach, a rule has to be declared in *all* monitored classes' class definitions for a rule to apply to several classes. The second approach suffers an additional drawback, i.e., the rule specified as an attribute value cannot be inherited to subclasses.

- **Rule processing in large OODBs.** Given a huge set of scientific data, as found in the genome domain, and a set of rules, we need to have an efficient pattern matching algorithm to find which rules have their condition parts satisfied. The most straightforward approach is to have *all* rules check the data set to find whether data is available to match their condition parts. However, this approach is very inefficient. One way to improve the efficiency is to (1) have a rule subscribe to the classes that it needs to watch (a subscription mechanism), i.e., the classes mentioned in its condition part, (2) have monitored objects in the subscribed classes send signals to relevant rules (event signaling), and (3) only have the relevant rules check their condition parts, when notified. The event signaling and the subscription mechanisms will be discussed in detail in Section 3.2.2. To further improve rule evaluation, we also adopt an incremental rule evaluation approach. That is we store information about which objects partially satisfy each predicate of a rule. This information is used to incrementally update the status of the satisfied rules to determine when a rule is ready to be fired.

The key features of rules in Crystal are:

- Crystal supports both *instance-level* rules and *class-level* rules. Instance-level rules are used to monitor objects for which the user has special interest. For example, a user may want to watch every condition associated with a particular gene, such as the breast cancer gene. Any update operation involving this gene should be brought to the immediate attention of the user. On the other hand, class-level rules are used to monitor conditions in a broader domain. For example, if we know the YACs on plate 3 are contaminated, we could specify a *class-level* rule as follows: "Ignore the experimental results that are derived by using YACs located on plate 3." This rule subscribes to the class that stores all the experimental results, instead to a particular object in that class.

- Furthermore, Crystal also supports for rules to subscribe to *workspaces*, i.e., arbitrary user-defined groups of data. There are two reasons for this functionality. First, the potentially huge data set contained in the database collected over many years of experimentation can be broken down into smaller and more meaningful workspaces. Thus an application user can focus her attention on a smaller set. The system will now run more efficiently, because the problem size is reduced to a more manageable size. For example, a user may want to focus the physical map assembler tool on the q-arm of Chromosome 17, thus ignoring data related to the p-arm of Chromosome 17 and any of the other chromosomes for that matter. The second reason is that a user can extract unreliable data from the initial data set, and apply rules only to the most plausible data set, thus getting more meaningful results. For example, if a user knows in advance that a piece of data is implausible, then she can mark the data as questionable and make it unavailable for inferencing. To make this possible, we introduce the notion of a dependency mechanism to notify rules about changes in other related objects. Dependency mechanism ideas are discussed in detail in Section 4.

- The scope of a single rule may span several classes, with each predicate watching a different class. For example, we could have a rule with two predicates as follows:

  ```
  IF (1) The experimental results are derived by using probe X, and
     (2) Probe X is unreliable
  THEN Mark the experimental results as questionable, and notify application user.
  ```

  As we can see from this example, predicate 1 monitors the ExperimentalResult class, and predicate 2 monitors the Probe class. The condition part is satisfied, if the probe bound to the variable X in predicate 1 is equal to the probe bound to X in predicate 2.

- A class-level rule may apply to a *local* class extent, a *global* class extent, or any combination of these two (because a rule may span several classes as described above) as its rule scope. When the rule scope is global, a rule is applicable not just to the anchor class itself, but also to all subclasses rooted at the anchor class. That is, the rule is *inherited* by all subclasses rooted at the anchor class. As shown in Figure 8, the OrientedOrderingRelationship class is a subclass of the Non-orientedOrderingRelationship class. We can specify a rule as follows "If an ordering relationship does not bear any orientation information, such as *contains* and *not-disjoint-1* relationships (Figure 2), then it can be added into any local orientation frame". This rule should apply only to the Non-orientedOrderingRelationship class, but not to its subclass, the OrientedOrderingRelationship class. Therefore, this rule should subscribe to the Non-orientedOrderingRelationship class and have *local* class extent as its rule scope. On the other hand, we could specify a transitivity rule (see also Section 2.2.3) on the Non-orientedOrderingRelationship class with *global* class entent as its rule scope, because orientability is not an important issue here. Therefore, this transitivity rule is inherited by its subclass.

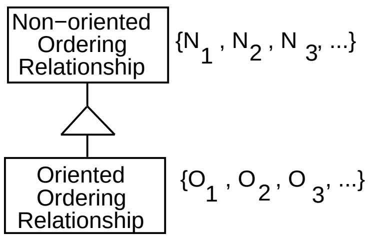
Non−oriented Ordering Relationship
{N1 , N2 , N 3, ...}
Oriented Ordering Relationship
{O1 , O2 , O 3, ...}

Figure 8: An Example Object Schema with Class Extents.

Because of the flexible modeling constructs of an OODB, we have the choice between the local and global extent[5] of a class as the scope of a predicate. Hence, Crystal is more expressive than the traditional rules typically found in Expert Systems, which deal with flat extremely simplistic data models.

- We want to support meta-rules, i.e., rules on rules. For this reason, we have designed rules that not only consume event signals from other objects, but also issue event signals to their own set of registered rules. Meta-rules allow us to make inferences about rules. For example, we could use a meta-rule for workspace implementation as follows "Only apply rules when the matched objects are in the same workspace".

- Rules can be grouped into meaningful collections, i.e., placed into particular rule groups. For example, we may have a set of rules that is relevant to the genetic map assembly task, and another set that is relevant to physical map assembly. Therefore, groupings of rules can be turned on and off to focus attention on particular subtasks, as desired. This is a key feature of a scientific database where different types of tasks must be accomplished on the same data sets at different times, including different degrees of ambiguity are acceptable depending on the stage of the experimental process.

3.2 Implementation of Crystal

To achieve these system objectives, we have built a layer of active system classes on top of the system class hierarchy of GemStone. The meta object model describing the most important active system classes given in Figure 9, they include: the ProductionRule, ProductionRuleSet Predicate, PredicateElement, AlphaSet, AlphaElement, ConflictSet, ConflictElement, ActiveObject, and ActiveSystem classes. This layered approach makes our system portable and generic. The ProductionRule class is used to support the storage, addition, modification, and deletion of the classical Condition-Action (CA) rules. It can subscribe to the classes that it needs to watch, and *react* to relevant event signals that are sent by these classes. The condition part of a rule is composed of zero or more Predicate objects, and each Predicate object in turn is composed of zero or more PredicateElement objects. An AlphaSet object is associated with each Predicate object to store the information used in the incremental rule evaluation process. Each rule then uses the information stored in AlphaSets to update its state, and the result, useful for rule execution, is stored in its ConflictSet object. A set of related rules can be held by a ProductionRuleSet object, and used for focusing attention on a specific task. The key function of the ActiveObject class is to send event signals to its subscribed rules. Therefore, all classes that are monitored by at least one rule are made subclasses of the ActiveObject class.

[5]Extent and extentAll are class methods provided by Crystal to model the local and global extents, respectively.

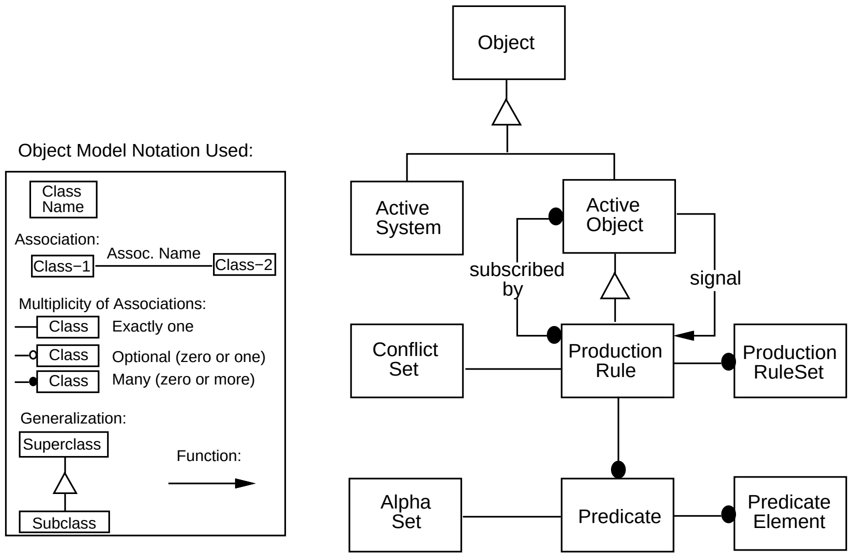

Figure 9: The Crystal Object Model (Simplified Version).

In the following subsections, we briefly discuss the key concepts of Crystal and indicate which system class (Figure 9) implements the concept.

3.2.1 Rule Modeling and Management

We treat rules as first class objects, and use the ProductionRule class to support the storage, addition, modification, and deletion of the classical Condition-Action (CA) rules. A CA rule contains a condition part and an action part. The condition part of a rule is composed of zero or more predicates, and each predicate is composed of zero or more predicate elements. Conditions have formal parameters that are passed from the rule's condition to the action code. The action code can be a database operation, for example, to create a new OrderingRelationship object, or an external operation, such as to notify the user that conflict data is found and to ask for user interaction. In our current system, actions are implemented in the data manipulation language (DML) of GemStone, i.e., in Opal. Rules can be grouped into sets, for instance a rule can be placed in zero or more ProductionRuleSet objects. While a rule is passive, it does not react to any event signals. The ProductionRule class has methods to activate rules. Rule activation is explained in Section 3.2.3.

Predicates, the building blocks of a rule's condition, are composed into more complex conditions. The predicates in a rule's condition are combined using logical AND operators[6]. Each predicate may contain zero or more predicate elements. Each predicate element consists of an attribute, an operator, and an operand (constant or variable). Each predicate has an associated alpha memory (an instance of the AlphaSet class) that records the results of all active objects that successfully match the predicate. An alpha memory element contains an object reference to the matching active objects and a collection of variable bindings necessary for the match to be true. Alpha memory is used in the incremental matching algorithm (Section 3.2.4). The parameters bound in a rule's condition part can be asserted in the action part. Each predicate has the following instance variables: type, scope, class, alphaMemory, and predicateElement. Type is used to indicate whether the predicate is positive or negative, and scope indicates whether we should use the local class extent or global class extent including all the extents of its subclasses. Class denotes the monitored active object class or instance.

[6]In our system, predicates using logical OR operators have to be broken into separate rules.

3.2.2 The Subscription and Notification Mechanisms

Next we discuss issues related to matching rules against large databases. With a potentially huge data set, it is inadequate to have *all* rules check the *whole* data set to find which rules have their condition parts satisfied. We discuss a more suitable way of processing rules in scientific databases.

The basic idea we utilize is to have the changed items in the database initiate the update process for the affected rules, rather than vice versa. To achieve this, we have designed a *subscription* and *event signaling* mechanism. A subscription mechanism can be viewed as a function as follows:

f(Class, Selector, Level, Timing, Scope) = { Rule },

where Class is a class, and Selector is a method defined in Class, Level indicates whether Selector is a instance or class level method, Timing indicates the event signal should be sent out *before* or *after* the method execution, and Scope indicates whether subscription is needed for instances of the local class alone or for the global class extent. When a method M of an ActiveObject instance is executed, the instance sends out an event signal to relevant rules, by searching the subscription list with the key value M upwards through the class hierarchy. The notification mechanism searches subscribed rules upwards, because there may exist a rule that subscribes to one of its superclasses with a global class extent as its rule scope. The upward searching process should stop once M is undefined in the class hierarchy. Therefore, the rule checking overhead is reduced to a minimal set, instead of forcing all rules check their condition parts at each cycle.

The ActiveObject class is a subclass of the standard Object class in GemStone that has been enhanced with a new active method subscription and event signaling mechanism. These mechanisms have been designed to notify ProductionRule objects whenever a method is executed on an ActiveObject instance. We provide ActiveObject instances with the capability of sending out event signals by introducing a new class method, called **`activeCompileAccessingMethodsFor`**, to the ActiveObject class. This method takes an array of instance attribute names and creates *active* versions of the read and write access methods for each instance attribute. These instance methods, generated automatically by our system upon creation of a new application-specific active class, encapsulate a pre-notification and a post-notification method call to check if there are any current subscriptions which are monitoring the access or setting of the attribute in question. If there are corresponding subscriptions, then the necessary signals are sent to the subscribing object(s) either before or after the method kernel code of accessing or setting internal data values is executed.

ProductionRule objects can register themselves to interested objects or classes, receive event signals from the objects, and react to the signals. Thus, event signals are not broadcasted to every rule, but to selective rules only.

3.2.3 Rule Activation

A rule can be in passive or activate state. When a rule is first activated, it computes the alpha memory sets, one for each of its predicates, then the conflict set that is used by the ActiveSystem object and by the incremental matching algorithm later on. The rule initialization stage is *passive*, because a rule does not react to or memorize any event signals while it is inactive. When a rule is first activated, its predicates use the information in their condition definitions to search the data set looking for data that satisfies the condition parts. That is, each predicate uses either the local class extent or the global class extent to build its alpha memory set. A rule also registers itself on the subscription lists of all its monitored instances or classes to assure it will be notified whenever these events occur from now onwards.

When all predicates of a rule are satisfied by objects in the database with consistent predicate variable bindings, we say this rule is satisfied. A satisfied rule is fired according to its priority. Because various rules may become true at the same time, the order of action execution must be decided by a rule's priority. While a default priority is set up for each rule, e.g., the total number of the predicate elements in its condition part, we provide the user with the capability to explicitly override the rule priority. A rule can have either immediate or deferred condition-action (CA) coupling mode. For the immediate CA coupling mode, whenever the rule's condition part is satisfied, the rule is fired right away. If it is a deferred, the rule is fired only at the time the transaction commits. Currently we found it sufficient to use the *coupled* rule-transaction mode where rule actions are fired as part of the user-defined transaction that triggered it.

3.2.4 The Incremental Rule Evaluation

Most rule systems exhibit a property called *temporal redundancy*. That is, the percentage of the changed objects to the total number of objects and predicates of the rules is small from cycle to cycle. That implies only a small percentage of rules are affected by the changes in the database. It is possible to take advantage of temporal redundancy to have a more efficient matching algorithm. There are several alternatives [20] that can make the overall matching process more efficient, such as providing condition membership (i.e., alpha memory size), memory support (i.e., alpha memory), and condition relationship (i.e., beta memory).

By remembering what has already matched from cycle to cycle and computing only the changes necessary for the newly asserted or retracted objects, unnecessary recomputation can be avoided. If only a small percentage of objects in the database is changed each cycle, then the matching process is fast. The costs of this more efficient incremental matching algorithm are that each rule must remember its partial matches. Among the different incremental matching algorithms, Rete [11] offers memory support and condition relationship aids. A major disadvantage of the Rete algorithm is that it uses a lot of memory space to store the partial match information. TREAT [20] is another incremental matching algorithm without the space storage penalty, because not all partial match information, in particular partial join results, are stored. TREAT provides memory support only. In this project, we have adopted a TREAT-like algorithm in the context of OODBs.

Crystal uses the incrementing matching algorithm, TREAT [20], to incrementally update its memory support. That is, after the initialization stage a rule checks its condition part only when subscribed events are sent to it. When a rule receives an event signal, it passes this message to its predicates. Each predicate checks whether the event is relevant to itself (Because a rule can subscribe to several classes, an event may be irrelevant to some predicates.) If the event is relevant to a predicate, the predicate checks whether the affected object satisfies its constraint and reports the results to the rule. Upon receiving the reports from its predicates, a rule calculates whether a set of objects and the corresponding variable bindings satisfy its condition. If true, then this information is added to a rule's conflict set and the rule is ready to be fired. As we can see, the affected object directs the pattern matching process, and only a set of selected rules react to the data modification.

3.2.5 Rule Execution

The rule execution provides the user a special type of transaction with two options to either commit the user-specified transaction and save the derived data or to commit the transaction without saving the derived data. The ActiveSystem class is designed for this purpose. It is a specialization of the standard System class provided by the GemStone OODB. Committing the user-defined transaction and saving the derived data is preferred, when the computational costs are high and we want to avoid recomputing data. It has however a potential problem of blowing up the data set. The second option could be taken when the computational costs are low.

When an ActiveSystem object is first started up, Crystal will activate all relevant rule objects associated with the ActiveSystem object and build a ConflictSet for each of the rule objects. An ActiveSystem object could have different sets of rule objects associated with it, stored in a ProductionRuleSet instance, and have them turn on and off different groups of rules at different times. This is a useful capability for experimenting with rule set orderings. More importantly, it also allows interested scientists to focus the system's activities on a designated task. Each ActiveSystem object keeps track of an agenda. The agenda contains the currently successfully matched rule objects ordered according to their priorities. The rule object with the highest priority will be fired first.

3.2.6 The Dependency Mechanism

To enable rules to subscribe to arbitrary user-defined workspaces, we have developed the following dependency mechanism. This is best explained via an example. For example, in Figure 10 user1's workspace contains 4 small collections, W_{or}, W_c, W_{yac}, and W_{sts}, of object instances extracted from OrderingRelationship, Chromosome, YAC, and STS class extents. User1 can have a set of rules, w1, w2, etc., monitoring methods executed on her selected workspace, instead of the entire class extents. Similarly, user2 may have different focus than user1, and she may thus select another set of collections as her current workspace. Although the create, delete, retrieve, and update methods executed on the OrderingRelationship class extent are relevant to {Rule c1, Rule c2, ...}, these events are irrelevant to {Rule w1, Rule w2, ...} unless the affected objects are also elements in the workspace W_{or}. On the other hand, the add and remove method events executed on workspace W_{or} are relevant to {Rule w1, Rule w2, ...} but irrelevant to {Rule c1, Rule c2, ...}. In general, create and delete events are relevant to rules associated with the class, while adding or removing workspace elements affect only rules subscribed to the workspace. Updating an object instance may have effects on both rules registered to the class and to the workspace. One simple way to allow rules to efficiently subscribe to a workspace is to modify our subscription mechanism as follows:

f(Class, Selector, Level, Timing, Scope) = { (Rule, Workspace) },

where Workspace indicates the region of interest. Before an ActiveObject instance generates an event signal to subscribed rules, it checks whether the event is executed on an object in the region of interest, i.e., in the workspace listed in the subscription. When a rule subscribes to a class extent, subscription lists a ϕ as the workspace parameter. If the object to which the event applies is not member of the indicated workspace parameter, then the event signaling process stops. This has been a very effective tool for reducing unnecessary event signaling, since the scientists typically focus their attention on a restricted region of data.

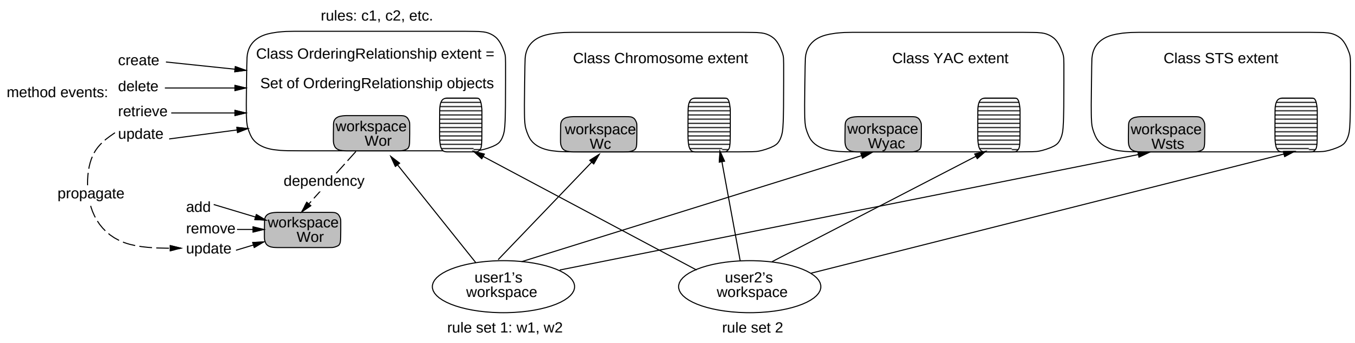

Figure 10: The Dependency Mechanism.

3.2.7 Crystal in Action

It is best to use an example to explain the rule processing procedures we designed. The OrderingRelationship class is made a subclass of the ActiveObject class, because it is watched by rules. In the example in Figure 11, each experimental data corresponds to an OrderingRelationship object. Assume the data set contains the OrderingRelationship object *before(A,B)* initially. Assume we add a new Rule object, Rule 1, into CA rule class. Rule 1, i.e., the transitivity rule, contains two predicates. When Rule 1 is activated, it computes AlphaSet for its predicates. In predicate 1, X is bound to A and Y is bound to B; and in predicate 2, Y is bound to A and Z is bound to B. Rule 1 is not satisfied, because the predicate variable bindings are not consistent. Therefore, nothing is placed into Rule 1's ConflictSet. When Rule 1 is activated, it also registers itself to the OrderingRelationship class. Rule 1 subscribes to the post-*create, delete*, and *update* events of the OrderingRelationship objects. When a new OrderingRelationship object, for example *before(B,C)*, is created, it sends the relevant event signals to Rule 1. Rule 1 then passes these messages to its predicates. Each predicate uses these messages to check whether the object satisfies its constraint, and reports the results to Rule 1. Rule 1 then uses this information to update its state. Now, the condition part of Rule 1 is satisfied, with X bound to A, Y bound to B, and Z bound to C, and the execution of the action part creates a new OrderingRelationship object *before(A,C)* using the variable bindings built in the condition part.

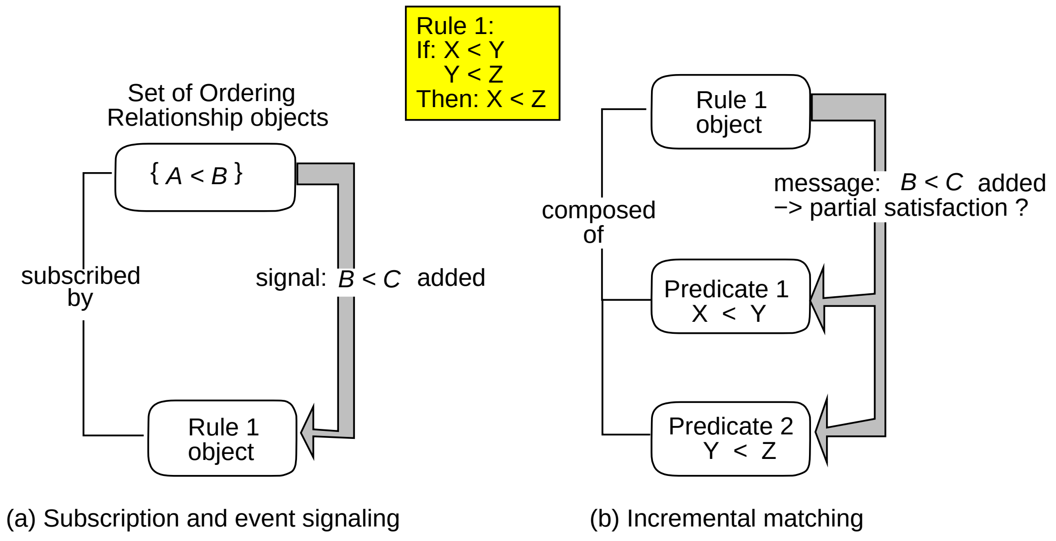

Figure 11: The Subscription Mechanism, Method Event Signaling, and Incremental Rule Evaluation.

4 THE PHYSICAL MAP ASSEMBLER SYSTEM

We have built the physical map assembler tool described in Section 2 on top of Crystal. In our system, we approach the physical map assembly task with the following procedure: given a set of pairwise ordering relationships between DNA fragments in many smaller local orientation frames, combine the local orientation frames into more informative (thus bigger) LOFs, compute the transitive closure, refine ordering relationships into more precise ones, then build physical maps using these derived data to span the whole chromosomes. It is straightforward to build this physical mapping application on top of Crystal, because Crystal provides the active system support for the inferencing capabilities. Therefore, the application user does not have to worry about the rule processing and event signaling issues. The resulting system forms a three-layered architecture system as shown in Figure 12. Each layer encapsulates the relevant functionalities for and hides unnecessary details from the next higher layer.

First, we have built a genome object model, as shown in Figure 1, that is capable of storing raw experimental data and the derived data and capturing the complex interrelationships between the objects. In particular, we have created the OrderingRelationship class in the genome object model (Figure 1) as a subclass of the ActiveObject class (Figure 9), because the inferencing rules react to method events generated by the OrderingRelationship instances. Other classes in Figure 1 are subclasses of the Object class.

Second, we have defined relationship refinement, LOF composition, and transitivity rules and have declared which classes in the object model are subclasses of the ActiveObject class. We have realized all relationship refinement, LOF composition, and transitivity rules introduced in Section 2.2 using the *parameterized* Crystal CA rule formats. Each rule thus corresponds to a separate rule object in Crystal. In the following, we will show some of the parameterized rules with the action part in pseudo code. For example, a parameterized LOF Composition Rule is specified as follows in Crystal:

```
if:
#( #( #pos #local #OrderingRel #( #( #type   #= #<rel1> )
                                   #( #arg1   #= #<A> )
                                   #( #arg2   #= #<B> )
                                   #( #LOFRef #= #<LOFi> )))
   #( #pos #local #OrderingRel #( #( #type   #= #<rel2> )
                                   #( #arg1   #= #<A> )
                                   #( #arg2   #= #<B> )
                                   #( #LOFRef #= #<LOFj> )
```

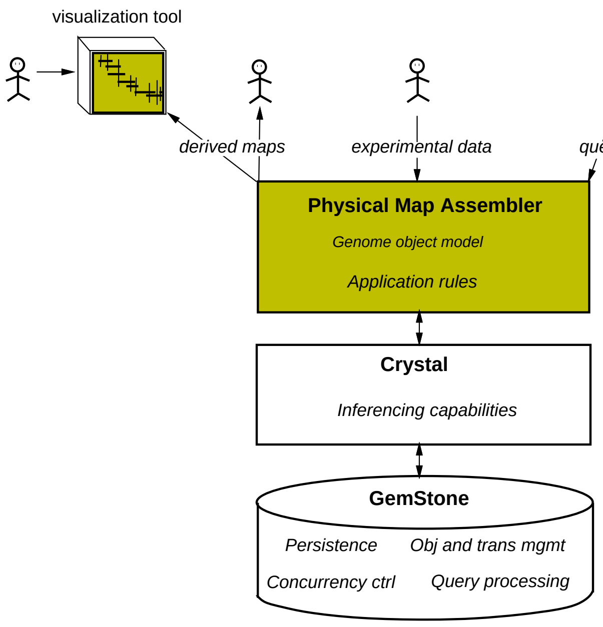

Figure 12: The Physical Map Assembler System.

```
                                   #( #LOFRef #~= #<LOFi> ))))
then:
[ action: IF no conflict is found based on rel1 and rel2
          THEN Insert a new ordering relationship, if applicable.
               Exchange arguments of the ordering relationships in LOFj, if needed.
               Combine LOFi with LOFj, if LOF composition is allowed.
          ELSE Report conflict data and turn control to the user ]
priority: 5
mode: deferred.
```

This rule has two predicates, and both predicates are positive. The scope of both predicates is local since this LOF composition rule is monitoring all the instances in the OrderingRelationship class extent. Predicate 1 has 4 predicate elements, and predicate 2 has 5 predicate elements. Therefore, the default priority of this rule is 9. The (user-defined) priority is 5. The executing mode is deferred. By setting the executing mode to be deferred, it allows the user to enter all the genomic input data first, then activate inferencing rules and fire rules at the end of this database transaction before the system is committed. The action part of the rule, stored in a block, is specified in the **then**-part. Because our inferencing rules monitor the extent of the OrderingRelationship class, the OrderingRelationship class is made a subclass of the ActiveObject class.

Similarly, a parameterized transitivity rule can be created in Crystal as follows:

```
if:
#( #( #pos #local #OrderingRel #( #( #type #= #<rel1> )
                                   #( #arg1 #= #<A> )
                                   #( #arg2 #= #<B> )
                                   #( #LOFRef #= #<LOFi> )))
   #( #pos #local #OrderingRel #( #( #type #= #<rel2> )
                                   #( #arg1 #= #<B> )
                                   #( #arg2 #= #<C> )
                                   #( #arg2 #~= #<A> )
                                   #( #LOFRef #= #<LOFi> ))))
then:
[ action: Insert an ordering relationship of interval A and C, if applicable;
          Do nothing, otherwise ]
priority: 1
```

```
    mode: deferred.
```

As shown above, genome rules are easily captured by the Crystal rule language. In our system, relationship refinement rules have a higher priority than the transitivity rules. Because each time a refinement rule is fired successfully, the total number of ordering relationships is guaranteed to be reduced. Hence, the overall complexity of the system (for example, the total number of ConflictSet elements that are calculated) is reduced accordingly. Similarly, LOF composition rules have a higher priority than the transitivity rules. Whenever a LOF composition rule is fired successfully, the total number of ordering relationships will be reduced at least by one. Transitivity rules have the lowest priority, because the total number of ordering relationships in the system may not be reduced after firing a transitivity rule.

The input to the system that we get from the biological scientists is an incidence matrix with the rows representing clones in a library and columns representing probes used in the experiments. The entries in the matrix indicate the reactions of the probes to the clones. We translate this matrix into our internal object representation, such as defining each probe as a Probe object, each clone as a Clone object, and each reaction as an OrderingReltionship object. The (potentially) huge data set can be reduced to a set of smaller, equal informative, and more manageable groups by putting clones with the same hybridization patterns into sets. For each such group we create a representative clone and analyze the data set through these representative clones. This is because every clone in the group has the same ordering and orientation relationships as the representative clone. The physical map assembler tool generates physical maps by applying inferencing rules to scientific data sets, and the maps can be viewed through a visualization tool. Although our physical map assembly tool can detect conflict data and notify the user, the system does not have a good conflict resolution strategy built in at this moment. Instead, the user uses the visualization tool to view the physical maps generated by our system to solve the conflicts manually. This suggestion about resolving conflicts can be fed back into the physical map assembler, e.g., by removing some conflicting overlaps from the current working set.

Finally, our physical map assembler is built on top of GemStone, and thus all database capabilities of GemStone are readily available. In particular, while GemStone does not provide a high-level query language, it provides rudimentary query mechanisms for searching over collections. The physical map assembler thus supports a set of pre-defined user queries, including "Give me all YACs on chromosome X" and "Give me all STSs that overlap YAC Y", to "Give me the most plausible map from the 1kb region of starting at q-arm of chromosome 17." Of course, other basic database functionalities, such as recovery, concurrency, etc,. are available for free.

5 EXPERIMENTAL SECTION

A Walk-Through Example First, we briefly describe one example of applying our approach to a physical map assembly problem, which has been successfully run by our Physical Map Assembler Tool. The goal is to demonstrate how our system can be utilized to get from the initial experimental data generated by laboratory experiments to the desired contig map. For this example we assume the following experimental data:

```
1.probe 1 hits YAC y1                3.probe 3 hits YACs y3, y4
2.probe 2 hits YACs y1, y2, y3       4.probe 4 hits YACs y4
```

Translation of experimental data into the object model representation: The experimental data is translated as follows: for each YAC y that is hit by a probe p create a *contains(y,p)* relationship object; for each YAC y in the target library that is not hit by the probe p create a *disjoint(y,p)* relationship object. Each containment constraint is placed into the global orientation frame object, while each disjoint constraint is placed into a new LOF. For example, the input from the third experiment will be translated into the following representation:

[*contains(y3,p3), contains(y4,p3)* in LOF_{global}], [*disjoint(p3,y1)* in LOF_i], [*disjoint(p3,y2)* in LOF_j].

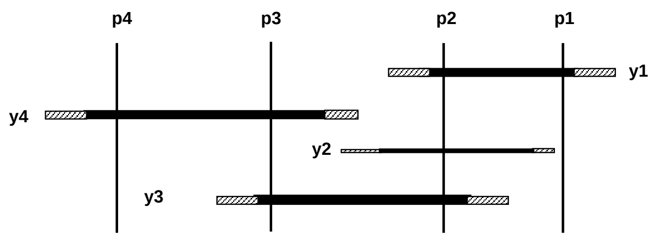

Figure 13: The Contig y4-y3-y1.

Ordering Inferencing: After we apply the inferencing rules to combine LOFs, compute the transitive closure, and refine the ordering relationships, as discussed in Section 2.2, our system will merge all relationships into one LOF. The most precise YAC-to-YAC ordering relationships derived by our system are:

ND<(y4,y3), O(y7,y4), NC2(y7,y3), NC2(y7,y1), O(y4,y1), and *ND>(y3,y1)* in a single LOF.

Map Generation: A physical map contains one or more ordered sets of DNA segments, while a contig is a special kind of map composed out of a set of *contiguous* overlapping clones. That is to say a map may contain more than one contig. To build the contig(s), we consider only relationships with known connectivity. In this example, we build the contig(s) using the following overlapping relationships: *ND<(y4,y3), O(y7,y4), O(y4,y1), ND<(y3,y1)*. Using the algorithm described in [18], we construct the contig "y4-y3-y1" depicted in Figure 13. The figure shows the final contig (the boldface lines) as well as the set of overlapping relationships generated by our system. The fragment y2 is not used in the contig, because it is covered by y1 ∪ y3.

An Example with Conflicting Ordering Relationships: Here, we briefly describe an example that leads to the discovery of conflicting ordering relationships. Our goal is threefold: (1) to show how we group probes with the same hybridization patterns[7] into a set and analyze the data using a representative probe from each group, (2) to demonstrate how our system detects conflicting data, and (3) to demonstrate how the visualization tool is used to allow the user to resolve the conflicts interactively, once discovered by our map assembly tool. Assume we have the following incidence matrix:

```
c1 30333000000000000
c2 03333333333333330
c3 00333333333333333
```

Each column of the incidence matrix represents a probe used in an experiment. As we can see from above, probe 1 and probe 2 have a unique hybridization pattern. However, probes 3, 4 and 5 have the same hybridization pattern[8], thus can be grouped together. We use probe P3 as the representative probe for this group. Similarly, probes p6 to p17 can be grouped into a set, and we use P4 as the representative probe for this group. After this data preprocessing, we can summarize our data set using the following format[9]:

```
1.probe P1 hits Cosmid  c1                4.probe P4 hits Cosmids c2, c3
2.probe P2 hits Cosmids c2                5.probe P5 hits Cosmid  c3
3.probe P3 hits Cosmids c1, c2, c3
```

Next, we translate the representative data into the object model representation, and then derive ordering information using the mechanism discussed in the previous example. This time our physical map assembler tool finds that the data set is inconsistent, because the inferencing process generates *disjoint(P2,P3)* and *disjoint(P3,P2)* in the same LOF when combining LOFs based on other ordering relationships. That is there is no consistent ordering of the probes that does not imply a deletion in one of the cosmids. Our visualization tool then displays the most plausible map[10] (Figure 14) to the user[11]. The user can easily see the conflict

[7] A column in the incidence matrix is called a hybridization pattern. For example, Column 1 of the incidence matrix is 300. It means P1 reacts to c1 positively, but reacts to c2 and c3 negatively.

[8] A column in the incidence matrix is called a hybridization pattern. For example, Column 1 of the incidence matrix is 300. It means P1 reacts to c1 positively, but reacts to c2 and c3 negatively.

[9] Here, we use c1 for C:148G9, c2 for C:116H2, and c3 for C:136D4 for simplicity's sake.

[10] The lines among the probes indicate the extent of the groups of probes. Each line is placed under and to the left of the LAST probe in the group.

[11] This figure is an actual screen dump generated by our visualization tool, developed in tk/tcl on Unix-based Sparc station.

graphically as displayed in Figure 14, as there is a broken line for cosmid C:148G9. Possible explanations include: (1) the experimental protocol (Alu/PCR amplification of Cosmid DNA followed by hybridization with cosmid grids) is subject to false negatives (most likely explanation), (2) one of the Cosmids has a deletion, or (3) one of the singular hybridizations is a false positive result. The user now can resolve the conflict shown in the picture by rearranging the order of the probes. In this example, for instance, the user could remove 140G10, the possible source that causes the inconsistency. Crystal then could run on the modified data set. Or otherwise, the user may suggest to repeat some experiments by using 140G10 (i.e., probe P2) against C:148G9 (i.e., c1) to find if the reaction is false negative, or using 148G9 (i.e., probe P1) against C:148G9 to check if the reaction is false positive.

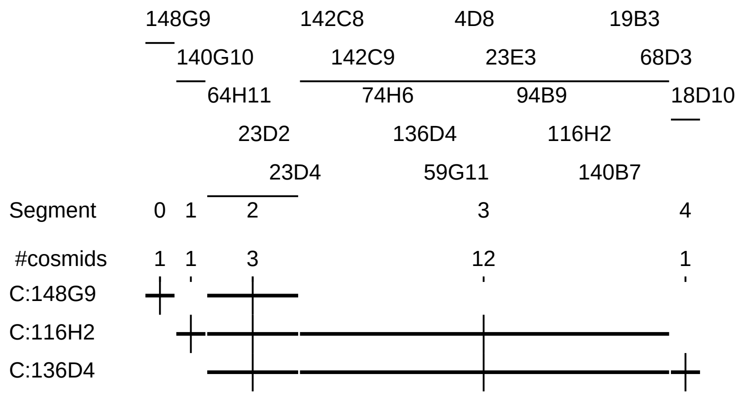

Figure 14: Visualization Output with Conflicting Data.

Other Results: In addition to the two examples we showed above, we have run numerous additional test cases. All test cases are run on the following platform: GemStone version 3.2.5 on a Sun Sparc 10 workstation (with 32 megs of memory) running SunOS 4.1.3. There are 10 parameterized rules currently in our physical map assembler (Section 2). The results are summarized in the following table:

Test	# of Frags	# of Rels (in)	# of LOFs (in)	# of Rels (out)	# of Rules Fired	Time (secs)
Test1	5	6	3	14	35	180
Test2	5	6	3	14	29	140
Test3	6	9	5	22	68	500
Test4	7	12	7	29	113	977
Walk-Thru	8	16	10	38	189	1807
Test5	9	20	13	47	259	3799

The second column of the table shows the number of input fragments, including YACs in the target library and probes used to run the experiments. The third column shows the number of input ordering relationships, and the fourth column shows the number of LOFs created at data entry time. In all test runs, the system was able to combine all input LOFs into a single LOF. Column 5 shows the number of the *most precise* ordering relationships generated by our system, the much larger number of less precise relationships are not generated by our system. Column 6 shows how many rules are fired in order to infer orderings and generate contig maps, and column 6 shows the CPU time each test case took. The ordering relationships generated by these test cases are identical to the results generated manually by the authors.

The first Crystal prototype, having been completed recently, has not yet been fine-tuned and further optimized. The time spent on each test case not only depends on how the rules are written, and the data sets we begin with, it also depends on the system load when the test cases are run. In order to improve the efficiency of our system, we plan to implement grouped attribute updates in Crystal. Currently, Crystal only generates events for single attribute updates. For example, if a new ordering relationship object *contains(YAC1,STS2)* in LOF$_{global}$ is created, this seamingly-simple database operation will generate one pre-creation event notification, one post-creation event notification, 4 pre-attribute-update event notifications, and 4 post-attribute-update event notifications to the subscribing rules (there are 4 attribute updates: one for the ordering type, one for argument 1, one for argument 2, and one for the LOF ID). With the aid of grouped attribute updates, we will be able to treat the operation as an atomic unit and issue one pre-grouped-update event notification and one post-grouped-update event notification - instead of 10 event notifications. This will considerably improve the performance of Crystal. We also are exploring mechanisms for relaxing the pattern matching process [3] to further improve performance.

6 RELATED WORK

6.1 Active OODBs

Active database research has become an increasingly important research area in the last few years. ADAM [8] and Sentinel [2] are the systems probably most closely related to Crystal. All three systems treat events and rules as first class objects. ADAM only supports a rule scope of classes, Sentinel and Crystal support both class and instance-level rule scopes. Crystal and Sentinel associate different meanings with the term "class-level" rules. In Sentinel, it means the scope of the rule is in one class. Hence, the rule may be defined in the class definition. In Crystal, a rule can monitor more than one class, and a rule can have local and/or global class extents as its rule scope. Therefore, we treat rules as independent objects, not implemented as part of a particular class. The concept of rules on workspaces is unique to our approach, probably driven by the need of the genomic domain to focus map analysis on particular subsets of data. In Sentinel, rules are ReactiveObject instances, but not ActiveObject instances (using our terminology). That is their rules are able to receive and record the events, but it appears that they are not capable of generating events. We made the ReactiveObject class a subclass of the ActiveObject class, thus rules (instances of the ReactiveObject class) are able to generate and consume event signals.

Currently, Crystal only supports primitive method events that are sufficient for production rule type of systems as required by our genomic application. Complex events, as reported by other systems [2], have not been addressed. Lastly, Crystal encapsulates unnecessary details of active system support from the application developer, whenever possible. Hence, event signaling in Crystal is transparent to the application users. For this purpose, we have developed a generic mechanism called activeCompileAccessingMethodsFor: that creates active versions of all attribute access methods for ActiveObject subclasses (Section 3.2.2). On the other hand, in Sentinel begin of message (bom) and end of message (eom) must be embedded in each class definition by the application user. Kotz-Dittrich[17] has also done experimental work on adding active functionality to GemStone. However, while our emphasis is on production rules, i.e., the processing of complex conditions, she has not addressed this issue. Her work do not provide any incremental rule matching support necessary for production rules, but rather focus on complex event processes.

6.2 Genome Physical Mapping Tools

To the best of our knowledge, Crystal is the first work in applying *active OODB* technology to genomic applications. There is on-going research, however, on developing object models to capture complex genetic data types. For example, Goodman [14] also advocates utilizing OODB technology for the informatics support of genome mapping projects. The focus of Goodman's work [14] is on genetic rather than physical map construction. In [16], Honda et al. describe an object model for genome data that covers several levels of resolution, including genome maps, DNA sequences, and gene sequences. Our work instead focus on providing a minimal genome object model as necessary to achieve the physical mapping task, rather than creating a generic all encompassing model.

When commercial database technology (i.e., without inferencing capabilities or relational technology) is used, software aiding genetic tasks, such as the construction of maps, typically are implemented as independent programs that must be run separately. They generally get and store their data in special-purpose data files, rather that directly integrating with an OODB [21]. On the other hand, the results of our system are objects directly stored in the database, hence they can be shared by users.

Ordering and orientation of the relationships among DNA fragments are two essential concepts in our project. We find the work of Allen [1] and Letovsky and Berlyn [19] closely related at this aspect. Allen [1] lists all possible relationships between temporal intervals assuming complete information about their durations, and presents a transitivity table for these relationships. The main difference between Allen's work and our work is that the ordering of temporal intervals is always global, while the ordering in our genome domain is relative to some local reference point. Letovsky and Berlyn [19] use local ordering windows to represent such relative orderings. Their work deals only with points rather than intervals. We combine the 'interval' concept [1] and the 'window' idea [19] to solve the DNA fragment ordering problem.

ICRF contig programs [21] have used heuristic or optimization approaches for ordering clone libraries. We use a rule-based approach to establish possible orderings, generate physical maps, and react to new insertions of experimental data. Since the mapping process is not yet well-understood, we elicit typical actions done by scientists and express them in the form of rules. This thus is an incremental process, as we can dynamically extend other capabilities of our system as new technology for generating maps get known.

SIGMA from LANL [25] is a system for generalized genome map assembly built using ObjectStore. It has several disadvantages from our point of view such as: the system uses compiled-in methods, thus changes to the algorithm require code rewriting. Unlike rules, these methods and their respective interactions cannot be as easily added, modified, and deleted dynamically. Other systems for storing and displaying genomic information are mostly, or completely, oriented towards display of generally static data. ACEDB[9], for example, has very good display modes for map information. However, it appears to have no map analysis capabilities.

7 CONCLUSION AND FUTURE WORK

We have developed a genomic object model that supports the efficient representation of both precisely and partially known ordering data, integrating both experimental and derived data for support of the physical map assembly task. We have also defined a set of inferencing rules that is required for deriving new ordering information from the initial data set. Our system allows incremental development of the data base as new experimental data is added, and provides for the identification of inconsistent information in ordering relationships and the potential of resolution of such situations. A key characteristic that distinguishes our physical map assembler from other approaches, such as [9] and [21], is its use of a rule-based mechanism and its smooth integration with OODB capabilities.

We have demonstrated that our approach is feasible by building a prototype of the physical mapping tool using Crystal. Because (1) Crystal supports the effective coupling of a database with a rule system and (2) the physical map assembler is built on top of Crystal, the analysis of the data, the derivation of physical maps, and reaction to new insertions of experimental data can be directly supported within the context of our unified active database system. This is the first time that active database technology has been applied to solve genomic problems. We have also shown how our approach can be used to effectively support the physical contig assembly task by providing a walk-through example and several experimental results.

We have identified that this scientific application needs both inferencing and complex object modeling. To meet this need, we have designed and implemented a seamlessly integrated active OODB system, called Crystal, that supports rule inferencing, complex object modeling, and typical database capabilities, such as persistence, concurrency control, and query processing. A key feature of Crystal is that events and rules are persistent objects. Hence, they can be added, modified, and deleted dynamically. Second, with the support of the subscription mechanism and method event signaling, the condition matching process is done incrementally, instead of re-executing all rules on the potentially huge data set. Also the system becomes truly active, i.e., it responds to changes in the data set with the relevant set of rules rather than having to check all the rules in the database. Third, because the ReactiveObject class is a subclass of the ActiveObject class, rules can both issue and receive events. Hence, Crystal supports meta-rules. Forth, rules can have local, global class extents, and any user-defined collections as scopes. This is an important feature of our system allowing scientists to focus on region of interest.

For the near future, we plan to focus on improving the efficiency of our first prototype, Crystal. For example, we will investigate better matching and indexing strategies. The dependency mechanism for monitoring changes to complex objects will also have to be reexamined. A key feature of our system for scientists is the workspace concept, which we plan to extend to deal with the generation and retraction of several plausible maps. We also plan to add a certainty propagation and conflict resolution mechanism as well as new types of inference rules, e.g., concerning distance, to our system. For acceptance and usage of our system directly by the scientists we also need to further refine our graphical user interface by more closely coupling the visualization support.

Acknowledgement. We would like to thank Douglas L. Moore for his help on implementing Crystal. Without him, Crystal would not be running today.

References

[1] Allen, J. F., "Maintaining Knowledge About Temporal Intervals", CACM, November 1983, Volume 26, pp 832-843.

[2] Anwar, E., Maugis, L., and Chakravarthy, S., "A New Perspective on Rule Support for Object-Oriented Databases", SIGMOD, 1993, pp. 99-108.

[3] Brant, D. A., Grose, T., Lofaso, B, and Miranker, D. P., "Effects of Database Size on Rule System Performance: Five Case Studies," In Proc. of VLDB, Barcelona, Sept. 1991, pp. 287-296.

[4] Carrano, A. V. et al., "Constructing Chromosome- And Region-Specific Cosmid Maps Of The Human Genome", Genome, 31, pp. 1059-1065, 1989.

[5] Cattell, R. G. G.,"Object data management : object-oriented and extended relational database systems", Addison-Wesley, 1991.

[6] Cormen, T. H., Leiserson, F. E., and Rivest, R. L., "Introduction To Algorithms", pp 469-476, 1991.

[7] Dayal, U., and Widom, J., "Active Database Systems", Tutorial, 1992 ACM SIGMOD International Conference on Management of Data.

[8] Diaz, O., Paton, N., and Gray, P., "Rule Management in Object-Oriented Databases: A Unified Approach," In Proc. of VLDB, Barcelona, Sept. 1991.

[9] Durbin, R. and Tierry-Mieg, J., "A C Elegans Database," documentation from FTP distribution at NCBI.

[10] Evans, G. A., "Combinatoric Strategies For Genome Mapping", BioEssays, Vol. 13, No. 1, January 1991.

[11] Forgy C. L., "Rete: A Fast Algorithm For The Many Pattern/Many Object Pattern Match Problem", Artificial intelligence, 1982, pp 17-37.

[12] Freksa, C., "Temporal Reasoning Based On Semi-Intervals", Artificial Intelligence 54, pp 199-227, 1992.

[13] French, J. C. Jones, A. K., and Pfaltz, J. L., "Scientific Database Management", University of Virginia, Technical Report 90-21, Dept. of Computer Science, Aug. 1990.

[14] Frenkel, K. A., "The Human Genome Project and Informatics", CACM, Nov. 91, Vol. 34, No. 11, pp 41-51.

[15] Guidi, J. N. and T. H. Roderick, "Inference of Order in Genetic Systems," The First Internal. Conf. on Intelligent Systems for Molecular Biology, July 1993.

[16] Honda, S., Parrot, N. W., Smith, R., and Lawrence, C., "An Object Model for Genome Information at All Levels of Resolution," Proc. of 26th Annual Hawaii Internat. Conf. on System Sciences, Vol. I, pp. 564 - 573, 1993.

[17] Kotz-Dittrich, A., "Adding Active Functionality to an Object-Oriented Database System - a Layered Approach," Proc. GI Fachtagung, Datenbankensysteme in Buero, Technologie, and Wissenschaft, BTW 93, Braunschweig, 1993, Springer Verlag.

[18] Lee, A. J., Rundensteiner, E. A., Thomas, S., and Lafortune S., "An Information Model for Genome Map Representation and Assembly." ACM 2nd Int. Conf. on Information and Knowledge Management (CIKM '93), Nov. 1993.

[19] Letovsky, S. and Berlyn, M. B., "CPROP: A Rule-Based Program For Constructing Genetic Maps", Genomics 12, pp 435-446, 1992.

[20] Miranker, D. P., "TREAT: A New And Efficient Match Algorithm For AI Production Systems", Pitman Publishing, pp 13-47.

[21] Mott, R., Grigoriev, A., Maier, E., Hoheisel, J., and Lehrach, H., "Algorithms and Software Tools for Ordering Clone Libraries: Application to the Mapping of the Genome of Schizosaccharomyces Pombe", Nucleic Acids Research, 1993, Vol. 21, No. 8, pp. 1965-1974.

[22] Pecherer, R., "Contig Graph Tool: A Graphical Interface For Contig Physical Map Assembly", Hawaii Int. Conf. on System Sciences, Vol. I, pp. 544-553, 1993.

[23] Prerau, D. S., "Developing and Managing Expert Systems", Addison-Wesley, 1990.

[24] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., "Object-Oriented Modeling and Design", Prentice Hall, 1991.

[25] *SIGMA System for Integrated Genome Map Assembly, Version 0.70 User Manual*, Theoretical Bilolgy and Biophysics Group, Center for Human Genome Studies, Los Alamos National Laboratory, 1992.

A. Table of Frame Composition Inference Rules

LOFj LOFi	NC<(A,B)	NC>(A,B)	ND1(A,B)	NC2(A,B)	ND<(A,B)	ND>(A,B)	D(A,B)	O(A,B)	C(A,B)	C(B,A)
NC<(A,B)	No	LOFi=LOFj NC2	No NC< –> ND<	LOFi=LOFj NC2	No NC< –> ND<	LOFi=LOFj O	LOFi=LOFj D	LOFi=LOFj O	No NC< –> C	Conflict
NC>(A,B)	LOFi=LOFj NC2	No	No NC> –> ND>	LOFi=LOFj NC2	LOFi=LOFj O	No NC> –> ND>	LOFi=LOFj D	LOFi=LOFj O	Conflict	No NC< –> C
ND1(A,B)	No NC< –> ND<	No NC> –> ND>	No	No NC2 –> O	No	No	Conflict	No	No ND1 –> C	No ND1 –> C
NC2(A,B)	LOFi=LOFj NC2	LOFi=LOFj NC2	No NC2 –> O	LOFi=LOFj	LOFi=LOFj O	LOFi=LOFj O	LOFi=LOFj D	LOFi=LOFj O	Conflict	Conflict
ND<(A,B)	No NC< –> ND<	LOFi=LOFj O	No	LOFi=LOFj O	No	LOFi=LOFj O	Conflict	LOFi=LOFj O	No ND< –> C	Conflict
ND>(A,B)	LOFi=LOFj O	No NC> –> ND>	No	LOFi=LOFj O	LOFi=LOFj O	No	Conflict	LOFi=LOFj O	Conflict	No ND> –> C
D(A,B)	LOFi=LOFj D	LOFi=LOFj D	Conflict	LOFi=LOFj D	Conflict	Conflict	LOFi=LOFj	Conflict	Conflict	Conflict
O(A,B)	LOFi=LOFj O	LOFi=LOFj O	No	LOFi=LOFj O	LOFi=LOFj O	LOFi=LOFj O	Conflict	LOFi=LOFj	Conflict	Conflict
C(A,B)	No NC< –> C	Conflict	No ND1 –> C	Conflict	No ND< –> C	Conflict	Conflict	Conflict	No	Conflict
C(B,A)	Conflict	No NC> –> C	No ND1 –> C	Conflict	Conflict	No ND> –> C	Conflict	Conflict	Conflict	No

Note: For some cells even though LOFi and LOFj cannot be combined, one of the relationships can be upgraded to a more precise relationship.
If this is the case, the upgraded relationship is shown in the second entry.

B. Table of Transitivity Rules

rel2 rel1	NA	NC<	NC>	ND1	NC2	ND<	ND>	D	O	C

NA	no info
	no info
	no info
	no info
		no info
		no info
		no info
		NA(C,A)
			NA(A,C)
			no info
			no info
			no info
				no info
				no info
					NA(A,C)
					no info
					no info
					NA(C,A)
						no info
						no info
						no info
						NA(C,A)
							NA(A,C)
							no info
							no info
							no info
								NC<(A,C)
								no info
								no info
								NC>(C,A)
									NA(A,C)
									no info
									no info
									NA(C,A)
										no info
										NA(A,C)
										no info
										NA(C,A)

NC<	NA(A,C) no info no info no info	NC<(A,C) no info no info NC<(C,A)	NA(A,C) no info no info no info	NA(A,C) no info	NC<(A,C) no info no info NC<(C,A)	NC<(A,C) NA(A,C) no info NC<(C,A)	NA(A,C) NA(A,C) no info no info	NC<(A,C) no info no info D(C,A)	NC<(A,C) NA(A,C) no info NC<(C,A)	NC<(A,C) NA(A,C) no info NC<(C,A)

NC>	no info
	no info
	no info
	NA(C,A)
		no info
		no info
		no info
		NA(C,A)
			NC>(A,C)
			no info
			no info
			NC>(C,A)
				no info
				NA(C,A)
					NC>(A,C)
					no info
					no info
					NC>(C,A)
						no info
						no info
						NA(C,A)
						NA(C,A)
							NC>(A,C)
							NA(A,C)
							NA(C,A)
							NC>(C,A)
								D(A,C)
								no info
								no info
								NC>(C,A)
									NC>(A,C)
									NA(C,A)
									NA(C,A)
									NC>(C,A)
										no info
										NC>(A,C)
										NC>(C,A)
										NA(C,A)

ND1	no info
	no info
		no info
		NA(C,A)
			NB(A,C)
			no info
				no info	NA(A,C)
					NA(C,A)
						no info
						NA(C,A)
							NA(A,C)
							no info
								NC<(A,C)
								NC>(C,A)
									NA(A,C)
									NA(C,A)
										no info
										ND1(A,C)

NC2	NA(A,C)
	no info
	no info
	NA(C,A)
		NC<(A,C)
		no info
		no info
		NC<(C,A)
			NC>(A,C)
			no info
			no info
			NC>(C,A)
				NB(A,C)
				NB(C,A)
					NC2(A,C)
					no info
					no info
					NC2(C,A)
						NC<(A,C)
						NA(A,C)
						NA(C,A)
						NC<(C,A)
							NC>(A,C)
							NA(A,C)
							NA(C,A)
							NC>(C,A)
								D(A,C)
								no info
								no info
								D(C,A)
									NC2(A,C)
									NA(A,C)
									NA(C,A)
									NC2(C,A)
										NC<(A,C)
										NC>(A,C)
										NC>(C,A)
										NC<(C,A)

ND<	NA(A,C)
	no info
	no info
	no info
		NC<(A,C)
		NA(C,A)
		no info
		NC<(C,A)
			NA(A,C)
			no info
			NA(A,C)
			no info
				NB(A,C)
				no info
					NC<(A,C)
					NA(C,A)
					NA(A,C)
					NC<(C,A)
						NC<(A,C)
						ND1(A,C)
						no info
						NC<(C,A)
							NA(A,C)
							NA(A,C)
							NA(A,C)
							no info
								NC<(A,C)
								NC>(C,A)
								NC<(A,C)
								D(C,A)
									NC<(A,C)
									ND1(A,C)
									NA(A,C)
									NC<(C,A)
										NC<(A,C)
										ND1(A,C)
										no info
										ND<(C,A)

ND>	no info
	no info
	no info
	NA(C,A)
		no info
		NA(C,A)
		no info
		NA(C,A)
			NC>(A,C)
			no info
			NA(A,C)
			NC>(C,A)
				no info
				NB(C,A)
					NC>(A,C)
					NA(C,A)
					NA(A,C)
					NC>(C,A)
						no info
						NA(C,A)
						NA(C,A)
						NA(C,A)
							NC>(A,C)
							no info
							ND1(A,C)
							NC>(C,A)
								D(A,C)
								NC>(C,A)
								NC<(A,C)
								NC>(C,A)
									NC>(A,C)
									NA(C,A)
									ND1(A,C)
									NC>(C,A)
										no info
										ND>(A,C)
										NC>(C,A)
										ND1(A,C)

D	NC>(A,C)
	no info
	no info
	NC<(C,A)
		D(A,C)
		no info
		no info
		NC<(C,A)
			NC>(A,C)
			no info
			no info
			D(C,A)
				NC>(A,C)
				NC<(C,A)
					D(A,C)
					no info
					no info
					D(C,A)
						D(A,C)
						NC>(A,C)
						NC<(C,A)
						NC<(C,A)
							NC>(A,C)
							NC>(A,C)
							NC<(C,A)
							D(C,A)
								D(A,C)
								no info
								no info
								D(C,A)
									D(A,C)
									NC>(A,C)
									NC<(C,A)
									D(C,A)
										D(A,C)
										NC>(A,C)
										D(C,A)
										NC<(C,A)

O	NA(A,C)
	no info
	no info
	NA(C,A)
		NC<(A,C)
		NA(C,A)
		no info
		NC<(C,A)
			NC>(A,C)
			no info
			NA(A,C)
			NC>(C,A)
				NA(A,C)
				NA(C,A)
					NC2(A,C)
					NA(C,A)
					NA(A,C)
					NC2(C,A)
						NC<(A,C)
						ND1(A,C)
						NA(C,A)
						NC<(C,A)
							NC>(A,C)
							NA(A,C)
							ND1(A,C)
							NC>(C,A)
								D(A,C)
								NC>(C,A)
								NC<(A,C)
								D(C,A)
									NC2(A,C)
									ND1(A,C)
									ND1(A,C)
									NC2(C,A)
										NC<(A,C)
										ND>(A,C)
										NC>(C,A)
										ND<(C,A)

C	NA(A,C)
	NA(C,A)
	no info
	no info
		NC<(A,C)
		NA(C,A)
		no info
		NC<(C,A)
			NA(A,C)
			NC>(C,A)
			NC>(A,C)
			no info
				ND1(A,C)
				no info
					NC<(A,C)
					NC>(C,A)
					NC>(A,C)
					NC<(C,A)
						ND<(A,C)
						ND1(A,C)
						no info
						NC<(C,A)
							ND1(A,C)
							ND>(C,A)
							NC>(A,C)
							no info
								NC<(A,C)
								NC>(C,A)
								D(A,C)
								D(C,A)
									ND<(A,C)
									ND>(C,A)
									NC>(A,C)
									NC<(C,A)
										C(A,C)
										ND1(A,C)
										no info
										C(C,A)

In each cell,

entry 1 = rel1(A,B) and rel2(B,C) entry 3 = rel1(B,A) and rel2(B,C)
entry 2 = rel1(A,B) and rel2(C,B) entry 4 = rel1(B,A) and rel2(C,B)