TECHNICAL RESEARCH REPORT

ICON: A System for Implementing Constraints in Object-based Networks

by S. K. Goli, J. Haritsa, N. Roussopoulos

CSHCN T.R. 94-11
(ISR T.R. 94-62)

The Center for Satellite and Hybrid Communication Networks is a NASA-sponsored Commercial Space Center also supported by the Department of Defense (DOD), industry, the State of Maryland, the University of Maryland and the Institute for Systems Research. This document is a technical report in the CSHCN series originating at the University of Maryland.

Web site http://www.isr.umd.edu/CSHCN/

ICON: A System for Implementing Constraints in Object-based Networks

Shravan K. Goli
Dept. of Comp. Sc., and ISR, Univ. of Maryland, College Park.

Jayant Haritsa
IISC, Bangalore, India, and ISR, Univ. of Maryland, College Park.

Nick Roussopoulos
Dept. of Computer Science, ISR, and UMIACS, Univ. of Maryland, College Park .

Abstract

In today's Network Management scenario, the network operator's interface to the network is through a Management Information Base (MIB). The MIB stores all management related data such as configuration information, performance statistics, and trouble logs and so on. Configuration management, which is at the core of network management, is implemented through the MIB in a three step process: making updates to MIB data elements, checking the validity of the updates, propagating the effects of the updates to the network elements. While all three steps need to be executed efficiently for the MIB to serve its intended goal, the second step of checking update validity is especially important from the management viewpoint. For example, if an operator mistakenly configures a ninth port on an eight port card, it is essential that the MIB should both detect and prevent this error. Allowing such operations to go through would have adverse impact on the performance of the network (since it increases the network management traffic). Therefore, we focus primarily on the problem of checking the validity of updates to MIB data elements, which can be viewed as a specific instance of the general problem of *constraint management* in database systems. We introduce the design of ICON (Implementing Constraints in Object-based Networks), a proposed constraint management system. In ICON, constraints are expressed through *rules*. Each rule is composed of an *event*, a *condition*, and an *action*. Occurrence of the event triggers the rule, the condition is a boolean check, and the action is executed if the condition is satisfied. Rules and events are also treated as objects so that they can be created, modified, and deleted like other objects, thus providing a uniform view of rules and evnets in an OO context. The OO paradigm results in an extensible and a reusable system. To our knowledge, not much work has been done in this area and this paper would trigger further research in this area.

1 Introduction

In enterprise communication networks, the network operator's interface to the network is through a Management Information Base (MIB). The MIB stores all management-related data such as network and system configurations, accounting information, and trouble logs. Configuration management, which is at the core of network management, is implemented through the MIB in a three step process: making updates to MIB data elements, checking the validity of the updates, propagating the effects of the updates to network elements. While all three steps need to be executed efficiently for the MIB to serve its intended goal, the second step of checking update validity is especially important from the management viewpoint. For example, if an operator mistakenly configures a ninth port on an eight port card, it is essential that the MIB should both detect and prevent this error. Therefore, in this paper, we focus primarily on the problem of checking the validity of updates to MIB data elements, which can be viewed as a specific instance of the general problem of ***constraint management*** in database systems. In particular, we introduce the design of ICON (Implementing Constraints in Object-based Networks), a proposed constraint management system intended for use in the *object-based* MIB of the PES (Personal Earth Station) network, a proprietary product of Hughes Network Systems, Inc., Germantown, Maryland, U.S.A. We also discuss here the integration of ICON with the PES data model and show how the configuration constraints of the various PES modules would be represented in ICON. A simplified ICON system prototype has been developed and integrated with a graphical user interface.

The remainder of this paper is organized in the following fashion: In Section 2, we present examples that highlight the diversity of network management constraints. Then, in Section 3, we discuss the technical issues underlying constraint maintenance in object-oriented database systems. The design of ICON, our proposed constraint management system, is presented in Section 4. The integration of ICON with the PES data model is discussed in Section 5, while details of the prototype ICON implementation are described in Section 6. Finally, in Section 7, we conclude with a discussion of future research avenues.

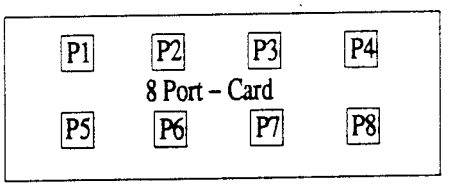
2 Examples of Constraints

A sample set of typical network management constraints is shown in Figure 1.

The constraint that attempting to configure more than 8 ports on an 8-port card exceeds the physical limitations of the card is expressed in Figure 1(a). Another type of constraint is shown in Figure 1(b) – here, the LAN type between communicating HUB and REMOTE LANs should be the same, that is, they should both be ethernet or both be token ring. In Figure1(c), it is mandated that the only legal values for a modem's baud rate attribute are 2400, 4800 and 9600. Finally, Figure1(d) states that only certain operators are allowed to make updates to parameters of network switches.

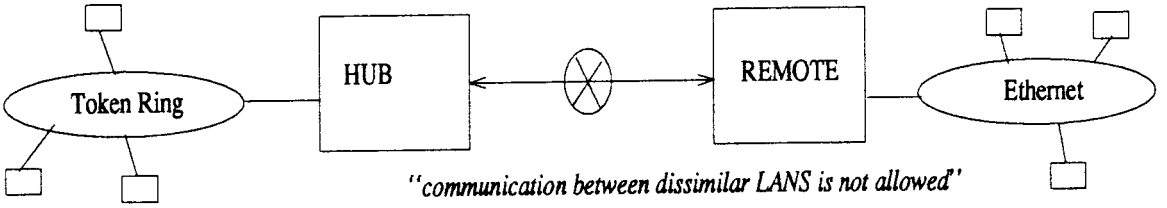
From the above examples, we observe that network management constraints have a variety of dimensions:

1. Constraints may be *physical* as in Figure 1(a), or *logical* as in Figure 1b.
2. Constraints may refer to a *single* object as in Figure 1(a) or span *multiple* objects as in Figure 1(b).
3. Constraints may be checked *immediately*, that is, as soon as the update is made, as in Figure 1(c) or *deferred* to a later time (e.g. completion of a related set of updates), as in Figure 1(b).
4. Constraints may apply *universally* to all applications accessing an object, as in Figure 1(c), or be *selectively* enforced based on the application accessing the object, as in Figure 1(d).



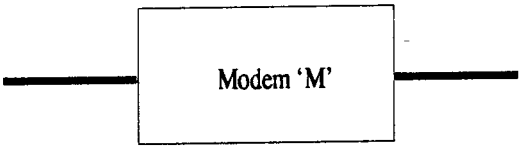
" no more than 8 ports can be configured on this card"

(a)



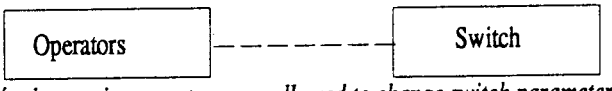
"communication between dissimilar LANS is not allowed"

(b)



"the legal baud rates are 2400, 4800, and 9600"

(c)



"only certain operators are allowed to change switch parameters"

(d)

Figure 1: Constraint Examples

3 Incorporating Constraints in OO DBMSs

Constraint maintenance in object–oriented databases differs from that of relational databases in many aspects, as discussed in [2, 3, 7, 12] . First, the constraints in object–oriented databases can be associated with specific objects, effectively localizing (or distributing) the constraints to individual objects. This eliminates the necessity of centralized control for rule maintenance. Second, different constraints can be associated with different instances of the same class. Third, relational databases support only some specialized forms of constraints like referential integrity constraints, security constraints, etc. Other types of complex constraints such as "an X.25 port cannot be connected to a SDLC port", are not easily supported. Such constraints can be supported only by embedding SQL statements in high level languages like C, PASCAL etc., but we then encounter the well-known problems of "impedance mismatch" [18] . A more detailed discussion of the differences between relational and object-oriented database management systems with respect to constraint management is given in [14].

Virtually all research into constraint management in object-oriented database systems has assumed that the constraints are expressed in the form of *rules*. Several OO systems that support such rules are described in the literature. For example, in [7, 12], rule management for Ode, a product of AT&T, has been described. In Ode, constraints are associated with class definitions. Ode has extended C++ [16] to O++ which provides facilities for associating constraints with an object. The specified constraints are checked every time an instance of that class is updated, a new instance is created, or an old instance is removed. Each constraint is expressed as a two-tuple <condition, action>. Whenever the constraint condition is violated, the action code is executed and the constraint is again tested. Detailed examples of how to express network management constraints in Ode are given in [14].

A different approach to constraint management, called ADAM, is described in [6]. Unlike Ode, where constraints are specified as a part of the object definition, the rules (or constraints) in ADAM are also treated as objects similar to the other objects in the system. Relationships can be established between the monitored objects and their associated rules. Each rule object maintains a list of monitored objects, and at the same time, the monitored objects also maintain a list of rules on them thus forming a two–way–relationship. In [14], a few examples of how to express network management constraints in ADAM are provided.

Yet another approach to constraint management, called Sentinel, is described in [2, 3]. The Sentinel approach captures the advantages of both Ode and ADAM, and extends them to provide significantly new features. It supports both constraints specified along with class definitions (as in Ode) as well as constraints specified as separate objects (as in ADAM). This has features to build rules spanning multiple objects which is difficult to do in both Ode and ADAM. In the following section, we discuss how several features of Sentinel were used in building the ICON system.

Apart from the aforementioned systems, mechanisms to express constraints in database systems are also given in [4] for O_2, in [13] for OOPS, and in [11] for CACTIS.

4 The Design of ICON

In this section, we present the design of the ICON network constraint management system, designed for use in the object-based MIB of the PES network (a satellite based network), a proprietary product of Hughes Network Systems, Germantown, Maryland, USA. The ICON design has taken most of its features from Sentinel (discussed in the previous section) and adapted them for the special requirements of the network management domain. The integration of ICON with the PES data model is described in Section

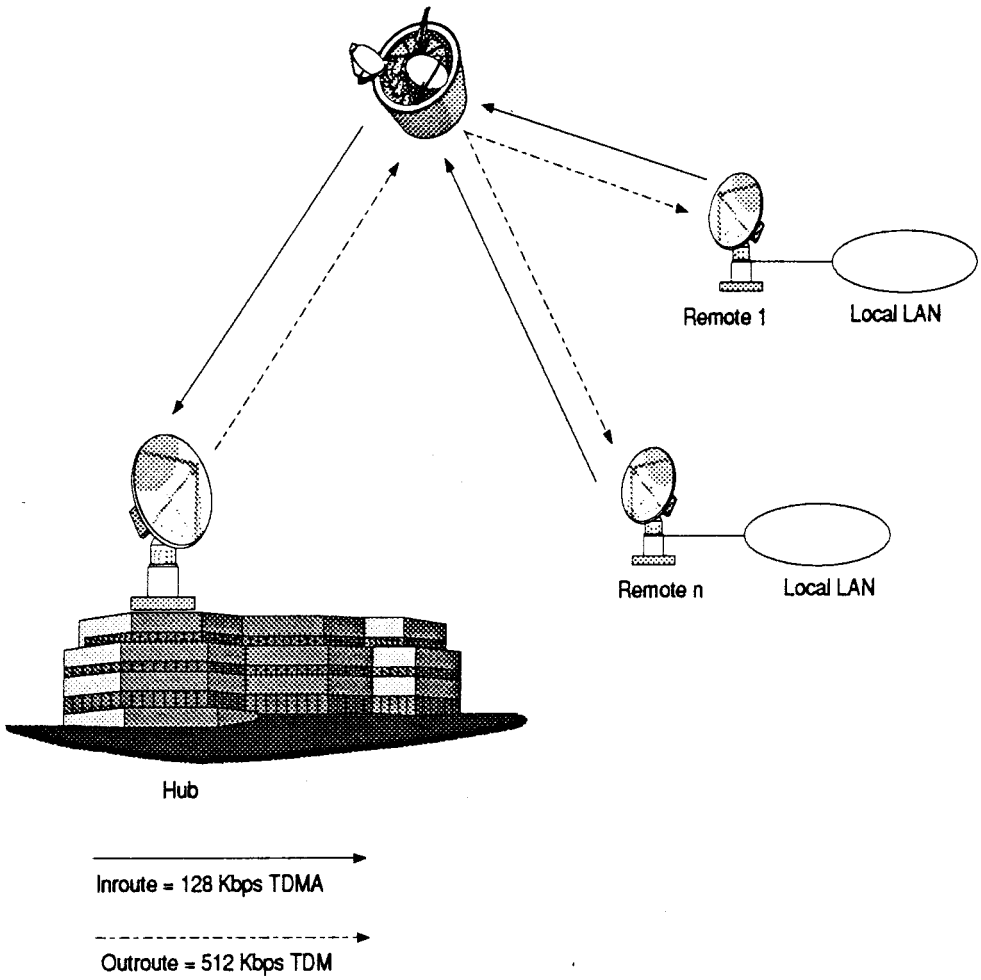


Figure 2: PES network

5, while the implementation details of the ICON prototype are presented in Section 7.

4.1 PES

In this section we give a brief description of the PES network. This network is composed of a hub, the systems control center and multiple remotes as shown in Figure 2.

- The hub provides centralized communication management for the remotes. All traffic between the remotes must pass through the hub; traffic cannot be passed from one remote to another directly over the satellite link.

- The systems control center (SCC) controls the network, i.e. all management of this network occurs from the SCC which is thus conceptually centralized but may be distributed in practice. The SCC and Hub is usually co-located. Management is done through operator consoles through which operators configure, monitor and control the network.

- The remotes are geographically dispersed sites that contain *remote node equipment.* The remote equipment is typically attached to customer equipment such as LANs, computers and workstations. Customer equipment is connected to the network via remote ports.

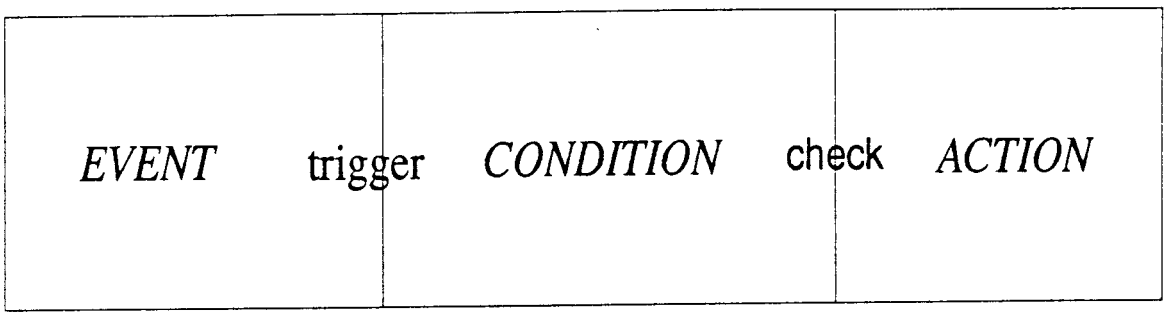


Figure 3: Rule Structure

Information is exchanged between the remote sites over a satellite link through the hub. Remote to hub transmissions travel over *inroutes*, while hub to remote transmissions travel over *outroutes* as shown in Figure 2. Thus, remote to remote travels from the remote to hub over an inroute, then from the hub to the other remote on an outroute. Of course, all transmission must be relayed through the satellite.

4.2 Design Details

We consider first the problems of how constraints are specified, how they are stored, and how they are evaluated and enforced. In ICON, constraints are expressed through *rules*. As shown in Figure 3, each rule is composed of an *event*, a *condition*, and an *action* (this is also known as the E-C-A paradigm [17]). Occurrence of the event triggers the rule, the condition is a boolean check, and the action is executed if the condition is satisfied. The above definition of rule is illustrated in the following example, which is used during configuration of the HUB module of the PES network to check for uniqueness of data port card (DPC) names:

```
class DPC{

  private:  char dpcname[20];
  public:   Set_Name(char *name);

  rule dpcname_uniq;
  when Set_Name(name)                    /* event */
     if not_unique(dpcname)              /* condition */
        then highlight(dpcname_field);   /* action */
```

In the above example, a class DPC is defined for data port cards. The rule dpcname_uniq monitors the configuration of DPC objects, and is triggered whenever a DPC object invokes the method Set_Name. A check is then made as to whether or not the 'dpcname' is unique. If the name is not unique then the action routine 'highlight()' is called to indicate the error to the operator, as shown in Figure 4.

It is important to note that the term event used here does not refer to network events, but refers to *database events*. In our object-oriented framework, database events consist primarily of *object method invocations*. With respect to configuration management, database events would mainly be initiated by operator actions. More generally, however, we expect that network event messages received by the MIB during network operation could lead to the generation of one or more database events.

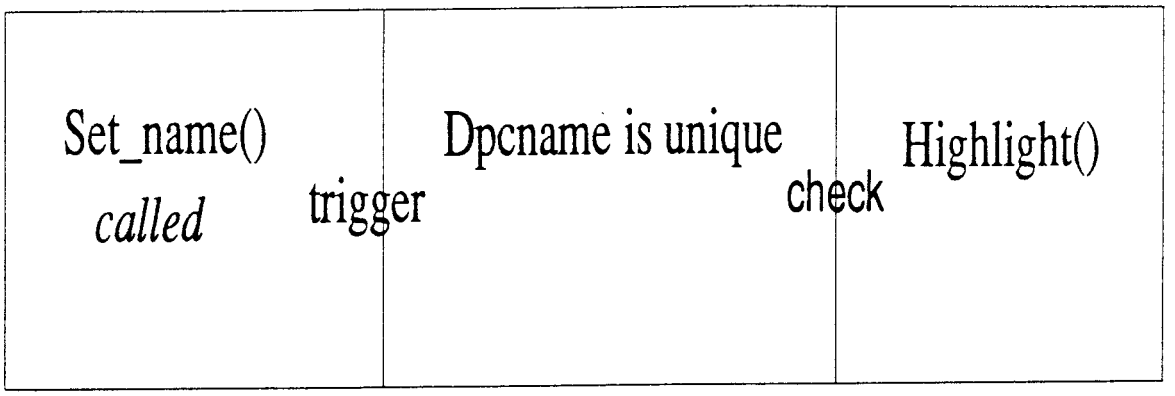


Figure 4: Rule Instance

4.3 Rule Specification

In ICON, rules are implemented as *first-class objects*, following the Sentinel approach. With this approach, rules can be created, modified, and deleted in the same manner as other objects, thus providing a uniform view of rules in an OO context. Second, rules are now separate entities that exist independently of other objects in the system. This allows for rule definitions to exist even when the object classes on which the constraints operate do not exist. Third, each rule has an object identity, thereby allowing rules to be associated with other objects. Finally, an extensible system is provided due to the ease of introducing new rule attributes or operations on rules.

4.4 Event Specification

In ICON, events are implemented as *first-class objects*, as in Sentinel. This implementation is chosen because events exhibit the properties of objects in terms of having state, structure and behavior. The state information associated with each event includes the occurrence of the event and the parameters computed when the event is raised. The structure of an event consists of the events it represents, while the behavior consists of specifying when to signal the event. By making events to be objects, events can be created, deleted, modified, and designated as persistent similar to other types of objects. The introduction of new event types and attributes is easily incorporated. Also, events spanning distinct classes can be expressed in a clean fashion. Complex events can be constructed using a hierarchy of event operators such as conjunction, disjunction, etc.

4.5 Event Generation

In ICON, each object is allowed to declare some subset of its public method interface to be *reactive*. This means that an event message is generated whenever a method in this reactive subset is invoked by the object. These event messages are propagated to other objects by a mechanism described in the following subsection. Event messages have the following structure:

Event Message = Oid + Class + Parameters

Here, *Oid* denotes the object identifier of the object generating the message, *Class* denotes the class of this object, and *Parameters* denotes the set of parameters with which the method is invoked.

4.6 Rule – Event Association

Rules are associated with events through a ***subscription*** mechanism. This mechanism allows rules to dynamically subscribe to the events generated by reactive objects. After the subscription takes place, the rule is informed whenever an object it subscribes to issues an event. Each reactive object maintains a list of its subscribed rules.

The above subscription mechanism results in some advantages: First, the runtime rule checking overhead is reduced since only those rules which have subscribed to an event are checked when that event is generated, that is, rule checking is localized (or distributed). Second, a rule is defined only once and it can be associated with any number of reactive objects. This is more efficient than defining the same rule multiple times and applying each rule to one type of object. Finally, rules triggered by events spanning distinct classes can be expressed. This is accomplished by a rule subscribing to the events generated by instances of different classes.

4.7 Object Classification

In ICON, objects are classified into the three categories described in Sentinel: passive, reactive, and notifiable. These categories and their relationship to events are described below.

Passive objects: These are regular C++ objects. They can invoke methods but do not generate events. Objects which do not need to be monitored fall into this category.

Reactive objects: Objects on which rules may be defined are made reactive objects. Once a method is declared as an event generator, its invocation will be propagated to other objects. Thus, reactive objects communicate with other objects via event generators.

Notifiable objects: Notifiable objects are those objects capable of being informed of the events generated by reactive objects. Therefore, notifiable objects become aware of a reactive object's state changes and can perform operations as a result of these changes. All rules are notifiable objects. There is an m:n relationship between reactive objects and notifiable objects, that is, a reactive object instance can propagate events to any number of notifiable object instances and a notifiable object instance can receive events from several reactive object instances.

4.8 ICON Example

The basic paradigm in ICON is that events are produced by reactive objects (producers) and they are consumed by notifiable objects (consumers). An example of Producer/Consumer paradigm is shown in Figure 5, taken from [2, 3]. Here, object P generates (produces) a primitive event event1 and sends it to a rule R1. The rule passes (consumes) the event to the event detector for storage and event detection, and if the event is detected, the rule checks the condition and takes appropriate actions. In this example, P is a reactive object, event1 is a primitive event, and R1 is a notifiable object. With reference to the earlier example in section 4.2, P is of type DPC and R1 is 'dpcname_uniq'.

4.9 Summary

The above design of ICON provides for: (i) rule definitions to be independent from the objects which they monitor, (ii) rules to be triggered by events spanning sets of objects, possibly from different classes, and (iii) objects to dynamically determine which object state changes they should react to and associate a rule object for reacting to those changes. Essentially, this separates the object and rule definitions from the event specification and detection process. This aids in building a modular and extensible system.

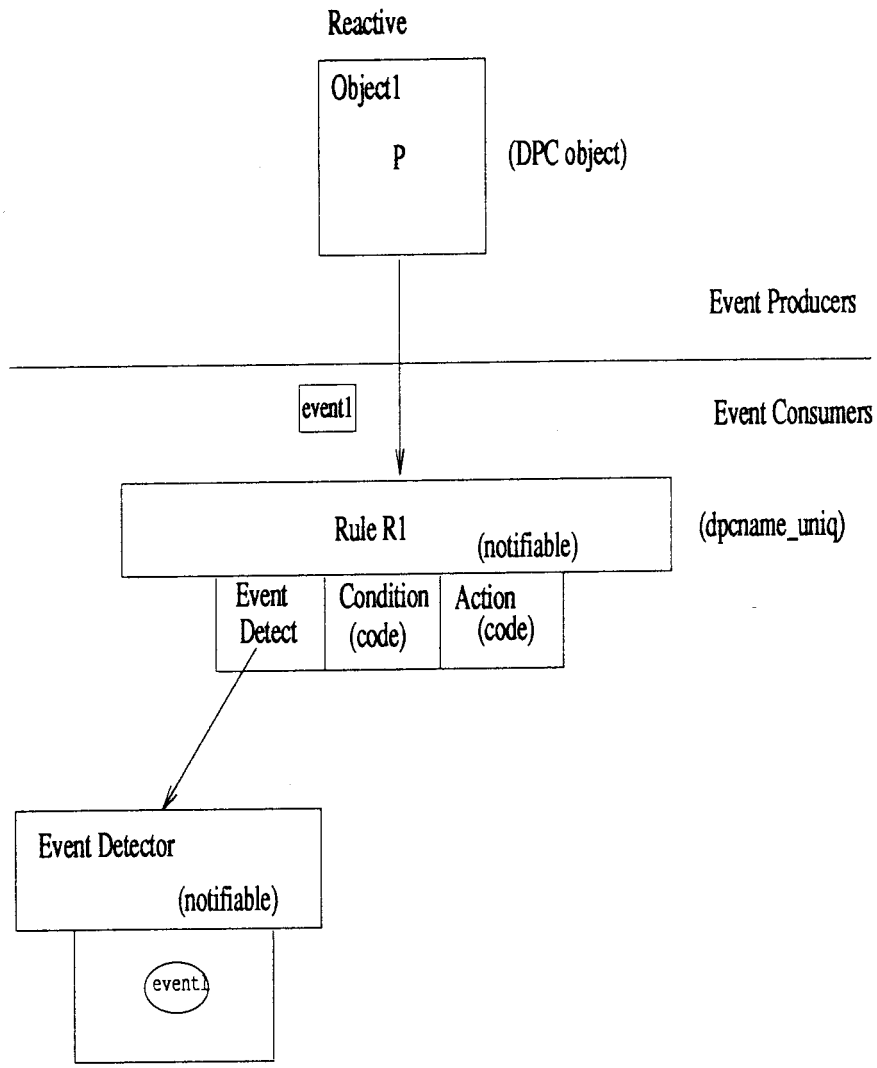


Figure 5: Event Producer/Consumer Analogy

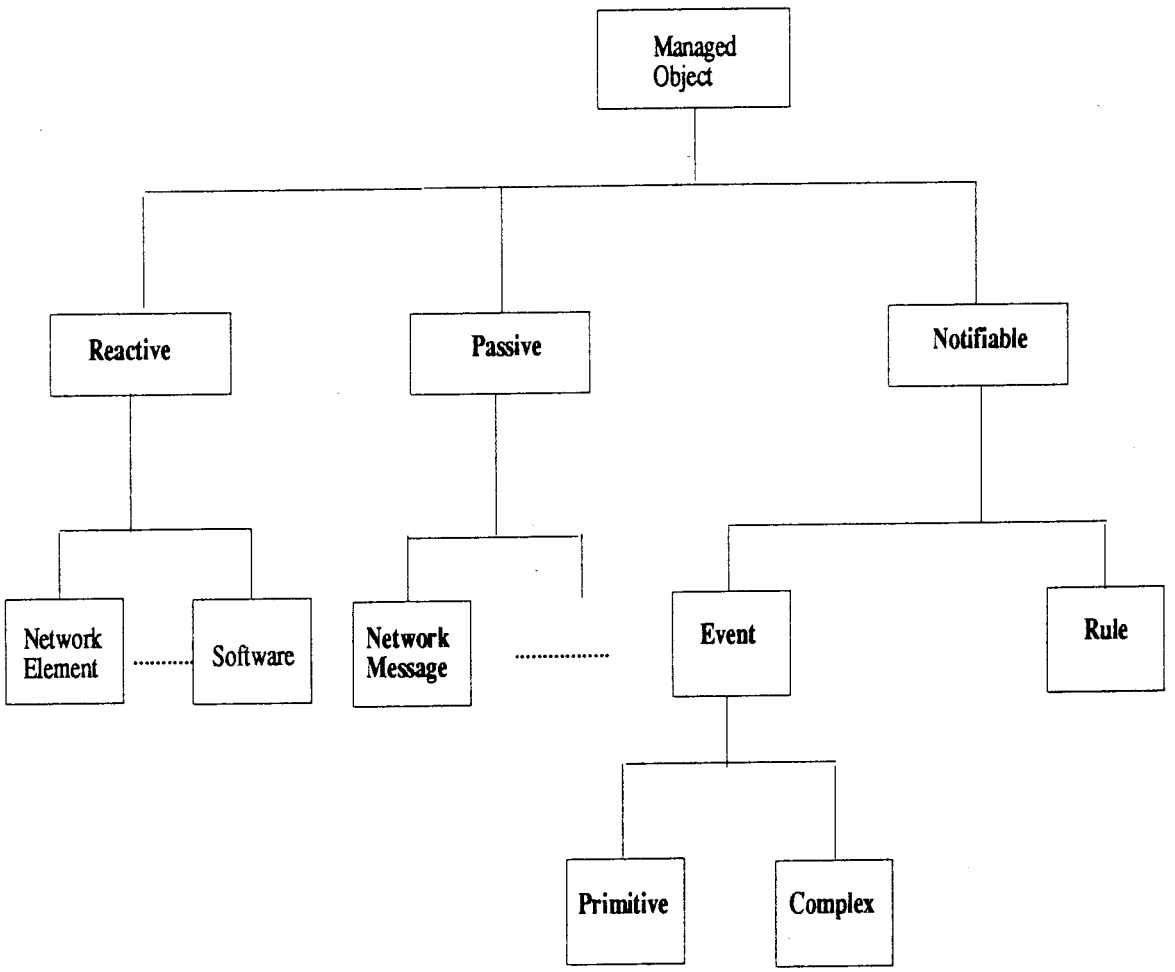


Figure 6: Integration with PES data model

5 Integration with Data Model

We now describe how the constraint management in ICON could be integrated with the PES data model. The details of the PES data model are given in [1]. For the purpose of integration, we have subdivided the top-level managed object class of the PES data model into a reactive class, a passive class, and a a notifiable class, as shown in Figure 6. All the elements of the PES data model which need to generate database events fall into the reactive class. Similarly, all the constraint management objects are in the notifiable subclass. The remaining objects, which do not generate any databse events or which do not have any rules imposed on them, fall into the category of passive objects.

6 Implementation details

As mentioned earlier, a prototype of a simplified version of ICON has been developed. In this section we discuss the implementation details of this prototype. The prototype was developed on the object-oriented database platform provided by ObjectStore, a commercial OO–DBMS [8, 9, 10].

We will first discuss the implementation of the Reactive, Notifiable, Event and Rule classes. Also, MOTIF/Galaxy [5] versions of graphical user interfaces were developed for specific examples.

6.1 Class Implementations

- **Reactive Class :** As shown in Figure 7, the Reactive class consists of protected data members *consumers* and *cons2b* . The *consumers* data member is the list of currently subscribed notifiable objects, and *cons2b* is the list of currently unsubscribed but relevant notifiable objects (i.e. consumers to be). The *Subscribe* method is used by a notifiable object to subscribe to a reactive object, while the *Unsubscribe* method removes a notifiable object from *consumers* and puts it in *cons2b* . Note that we could also have another method which would just unsubscribe the rules. Also, a method could be provided to directly update the *cons2b* list of rules. The *Reactive* method is simply a constructor. Notifiable objects are informed of events using the *Notify* method. This method forms the core of the ICON system and informs the consumers of : (i) the identity of the reactive object generating the primitive event, (ii) a unique string identifier that indicates the event generated, (iii) an error string which could be used to communicate the error messages, and (iv) the number and actual values of the parameters of the invoked method.

- **Notifiable and Event Classes :** The *Notifiable* class allows objects to receive and record primitive events generated by reactive objects. Both the Event class and the Rule class are subclasses of the Notifiable class; they receive and record primitive events generated by reactive objects. Notifiable class is presently a dummy class which is used to provide inheritance path.

  The event class hierarchy, which is taken from Sentinel [2, 3], is shown in Figure 8. The Event superclass provides the common structure and behaviour shared by all event types. The class schema for different kind of Events is shown in Figure 9. The Event class has a data member, *raised* , to denote the occurrence of the event. A method to reset the event is also provided. The *Primitive* class has attributes name and gen–obj. The gen–obj holds the address of the reactive object which generates the event. The complex event *And* is similarly defined; it gets raised when both the associated sub–events occur. An 'Or' event is similarly implemented. At the time of writing, the implementation for the 'And' event is as yet incomplete. The major problem here is that synchronizing the sub–events which occur asynchronously makes the implementation very difficult.

```
class Reactive : public  ManagedObject {

protected :
   os_Set<Notifiable*>  *consumers;          /* subscribed notifiable objects(rules) */
   os_Set<Notifiable*>  *cons2b;             /* unsubscribed notifiable objects(rules) */

public :
   Reactive();
   virtual ~Reactive();
   void Subscribe(Rule *obj);                /* allows subscription of new objects */
   void Unsubscribe(Rule *obj);              /* allows to unsubscribe a certain object */

   int Notify(Reactive *obj, char *event_name, char *err_str, int argc, ...);   /* notification method */

   static os_typespec*  get_os_typespec();   /* object store related definitions */
};
```

Figure 7: The Reactive Class

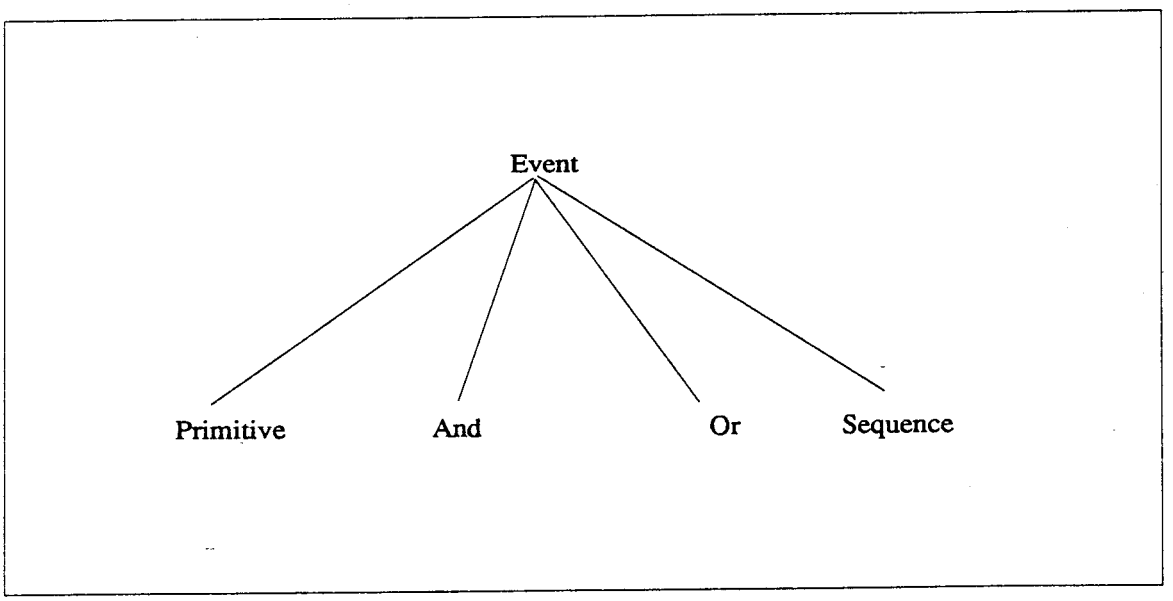


Figure 8: The Event Hierarchy

```
class Event : public Notifiable {

protected :
    int      raised;              /* event occurrence flag */

public :
   Event();
   virtual ~Event();
   virtual void event_off();   /* reset the raised flag */
   virtual int check_event(char *event_name);
   static os_typespec*      get_os_typespec();
};
```

```
class  Primitive : public Event {

protected :
    char  *event_name;              /* event name */
    Reactive *gen_obj;

public :
    Primitive(char *ev_name);
    virtual ~Primitive();
    int  check_event(char *ev_name);      /* check the name */
    static  os_typespec *   get_os_typespec();
};
```

```
class And : public Event {

protected :
    Event     *event_one, *event_two;  /* composition of 2 events */
     int          flag_one, flag_two;

public :
    And(Event *FirstEvent, Event *SecondEvent);
    virtual ~And();
    int  check_event(char *event_name);
    static os_typespec *      get_os_typespec();
};
```

Figure 9: Event Classes

```
class Rule : public Notifiable {

protected :
   char     *name;          /* rule name */
   int       enabled;       /* status flag */
   Event    *event_id;      /* event to react for */

public :
   int   Enable();
   int   Disable();
   void   Update(Event *eventid);
   virtual int Condition(Reactive *obj, char *err_str, int argc, void **argv);

   virtual int Action(Reactive *obj, char *err_srt, int argc, void **argv);

   int event_detect(char *name);
   int rule_stat();
   char *get_rule_name();

   Rule(char *in_name, Event *eventid, int in_enabled);
   ~Rule();

   static os_typespec * get_os_typespec();
 };
```

Figure 10: The Rule Class

- **Rule Class :** As shown in Figure 10, a rule is basically defined by the event which triggers the rule, the condition which is evaluated when the rule is triggered, and the result that is carried out if the condition is satisfied. Each rule object consists of data members *name, event−id* , and *enabled* . The attribute *name* takes as its value the name of the rule and can be used by the user to access the attributes and methods of the rule. The *event−id* attribute denotes the identity of the event object associated with the rule, and the attribute *enabled* denotes whether the rule is enabled or not. When a rule is enabled it receives and records propagated primitive events. The condition method is executed when the corresponding event occurs, and if satisfied, the action method is executed.

6.2 ICON Algorithm

We show here, in pseudo-code, a simple algorithm which describes the mechanism of event generation and rule checking in ICON:

```
algorithm_ICON()
{
      Whenever a Reactive method is accessed, at some point in its
      processing, a method called Notify() is used to send a message
      to all Rules subscribed to that reactive object;

      For each rule subscribed {
```

```
        if (the rule is enabled) {
          pass the event to the rule's event detector;
            if (event detected) {
              check the condition of the rule;
                if (condition is satisfied)
                perform the action;
          }
        }
      }

      return the appropriate value;

}
```

6.3 Graphical User Interface

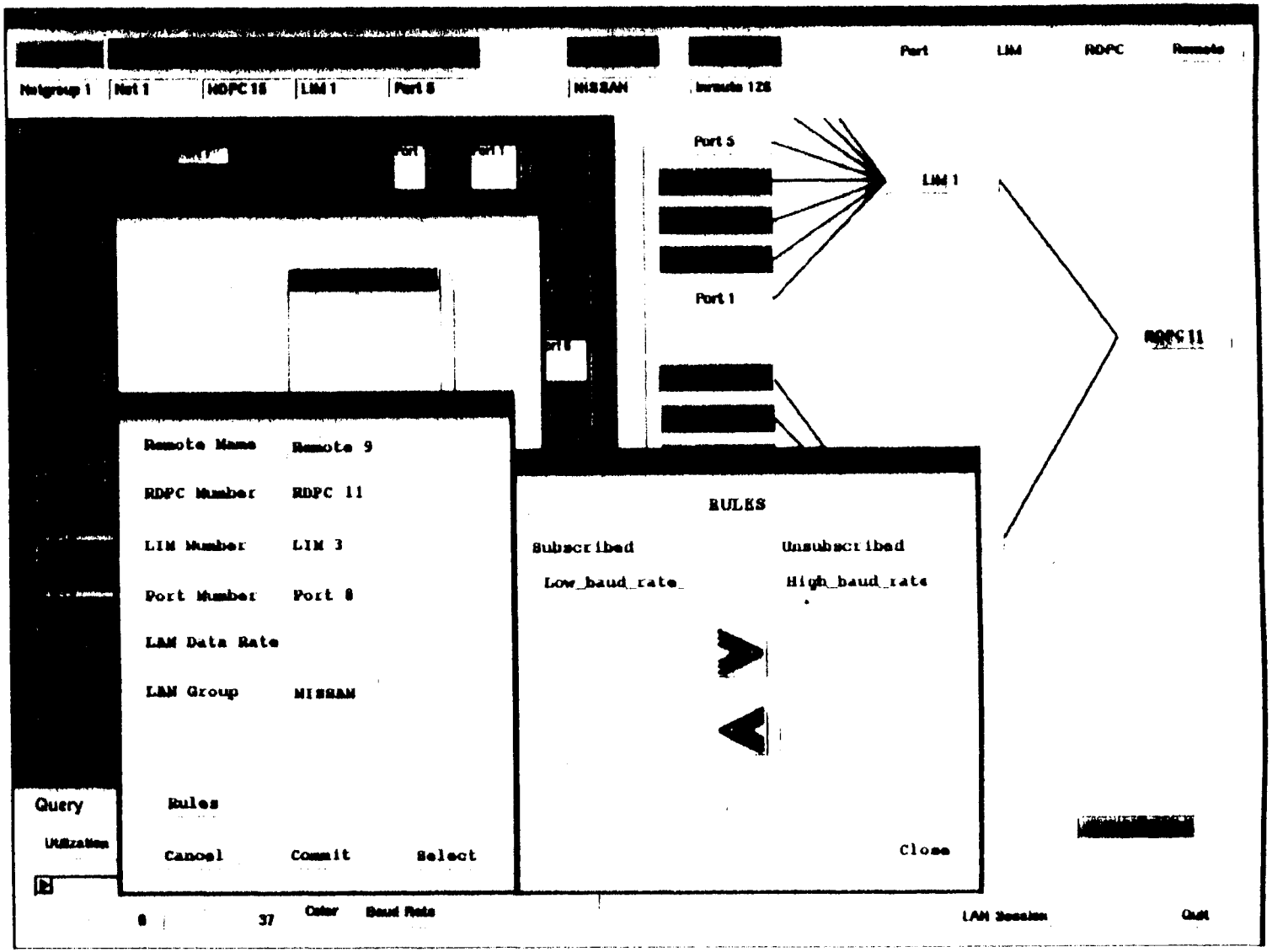
As described in earlier sections, rules exist as independent objects and they subscribe to reactive objects. Each reactive object maintains a list of currently subscribed rules, and a list of rules which are valid but currently unsubscribed (see section 6.1). It is possible to dynamically change these lists and, thereby, the behaviour of the monitored objects themselves. In our implementation, we have developed a Motif user interface to help the user of the system decide the behaviour of the monitored objects. The main advantage of this is that it helps in customization of the product to match different users requirements. Note that if the rules were implemented in the traditional way of writing code wherever required, this kind of dynamic behaviour would not have been possible. For any change, the code would have to be not only re–written but also recompiled. This would create several problems in network MIBs since recompilation requires database access to be suspended for other operations. In our mechanism, however, since rules exist independently, a user interface could be provided to assign new rules offline without interrupting any other operations.

In addition to the Motif interface, we have also developed a Galaxy version of the interface and integrated it with the Hub and Remote interfaces. Figure 11 shows an example of a Galaxy interface.

6.4 Implementation Examples

In this section, we give a few examples of constraint management using the constructs of ICON. An example in which a Session class is defined is shown in Figure 12. Whenever a session is being established between a hub port and a remote port, it is mandatory that both ports be connected to similar LAN types. Therefore, a rule to implement this constraint is defined, as shown in Figure 12. An instance of this rule along with the required event is shown in Figure 13. Whenever a new Session instance is created, the new object subscribes to this rule in the constructor method of the Session as shown in Figure 13. After this, whenever the actual data for this session instance is updated through the setup method, an event message is issued which would trigger the rule.

A real-code example of how configuration screens related to PES are implemented in ICON is shown in Appendix A. The constraint checking code given there is for the DPCDEF screen which is one of two screens used by the operator in configuring the HUB of the PES network. This code is virtually a direct translation of the existing code , which is written in Pascal. The translation was done into C++. Some



Unified interface for exploration, querying and data entry

Figure 11: Graphical User Interface example

```
class Session : public Link {
protected :
   char   DPCName[20];
   int      HubPortNum;
   char   RemName[20];
   int      RDPCPortNum;
   ..................
   ..................
public :
  Session(char *n=NULL);          /* general methods of Session class */
  void Setup(Network *hub, HDPC *hdpc, HDPC_LIM* hdpc_lim, Port* h_port,
             Remote *rem, RDPC *rdpc, RDPC_LIM* rdpc_lim, Port* r_port);
  void Fill();
};
```

```
Session_Porttype_Rule::Session_Porttype_Rule(char *in_name, Event *eventid,
      int in_enabled) : Rule(in_name, eventid, in_enabled) { }

                                        /* argc, argv are general variables - not from main */
int Session_Porttype_Rule::Condition(Reactive *obj, char *err_str, int argc, void **argv)
{
    if ( (((Port *)argv[0]->LAN) != (((Port *)argv[1]->LAN))
          return(TRUE);
    else return(FALSE);
}

int Session_Porttype_Rule::Action(Reactive *obj, char *err_str, int argc, void **argv)
{
   sprintf(err_str, "Port lan types dissimilar \n");
   return(ERROR);
}
```

Figure 12: Session/Rule definitions

```
Primitive *ev_session_port = new (db) Primitive("Session_Port_type_event");

session_port_rule = new(db) Session_Porttype_Rule("Session_Port_Rule", ev_session_port, 1)
```

```
Session::Session(char *n = Null) : Link(n)
{
  this->Subscribe(session_port_rule);
}

void Session::Setup(Network *hub, .....)          /* dots mean remaining args */
{
  char *err_str;
  if (Notify (this, "Session_Port_type_event", err_str_port, 2 , h_port, r_port)) {

        HubPortNum = h_port->GetID();
        RDPCPortNum = r_port->GetID();
        .....
        .....
  }
}
```

Figure 13: Usage of the defined Rule

more methods are required to be added to make it a complete example.

7 Research Issues

The design of ICON is still in its developmental stage. A number of research issues still remain open for investigation:

1. Currently, whenever an event is to be issued by a reactive object, it sends a message to all the rules associated with that event. An alternative approach would be for the reactive object to send a single message to a dynamic Notification Manager associated with that object, and for the Notification Manager to notify all the rules associated with the event. In this way, the load on the objects which generate events can be reduced. It may also be possible to perform some optimizations at the Notification Manager.

2. Maintenance of compatability between reactive and notifiable objects whenever new versions of either objects are introduced.

3. Mapping of inter-object constraints to appropriate intra-object constraints. This will result in efficient constraint checking.

4. Synchronization of sub-events of Complex events is a major research issue.

References

[1] A. Datta and M.O. Ball, "MOON :- A Data Model for Object Oriented Network management", to be published, ISR, University of Maryland at College Park.

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

[3] E. Anwar, "Supporting Complex Events and Rules in an OODBMS: A Seamless Approach", *Master's Thesis, Univ. of Florida* , 1992.

[4] C. Bauzer Medeiros, and P. Pfeffer, "A mechanism for Managing Rules in an Object-oriented Database", Altair Technical Report.

[5] C. Plaisant, H. Kumar, M. Teittinen, and B. Shneiderman, "Visual Information Management for Network Configuration", *to be pulished*, ISR, University of Maryland at College Park.

[6] O. Diaz, N. Paton, and P. Gray, "Rule Management in Object-Oriented Databases : A Unified Approach", in *Proc. of VLDB, Barcelona*, Sept 1991, 317-326.

[7] N.H. Gehani, and H.V. Jagadish, "Ode as an Active Database: Constraints and Triggers", in *Proc. of VLDB, Barcelona*, Sept 1991, 327-336.

[8] User Guide, ObjectStore Release 2.0, Object Design, Inc.

[9] Reference Manual, ObjectStore Release 2.0, Object Design, Inc.

[10] Class Notes, ObjectStore , Object Design, Inc.

[11] S.E. Hudson, and R. King, "CACTIS: A Database System for Specif ying Functionally-Defined Data", in *Readings in Object-oriented Database Systems*, eds. S.B. Zdonik, and D. Maier, Morgan Kaufmann Publishers, Inc., 1990 , 432-443.

[12] H.V. Jagadish, and X. Qian, "Integrity Maintenance in an Object-Oriented Database", in *Proc. of VLDB, Vanouver*, Aug. 1992, 469-480.

[13] G. Schlageter, R. Unland, W. Wikes, R. Zeischang, G. Maul, M. Nagl and R. Meyer, "OOPS - An Object Oriented Programming System with Integrated Data Management Facility", in *Proc. IEEE 4th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1988, 118-125.

[14] Shravan Goli, Jayant Haritsa, Nick Roussopoulos, "Integrity Constraints in Configuration Management", in *TR-xxx under preparation*, ISR, University of Maryland at College Park.

[15] Shravan Goli, Jayant Haritsa, Nick Roussopoulos, "A System for Implementing Constraints in Object-based Networks", in *TR-xxx under preparation*, ISR, University of Maryland at College Park.

[16] B. Stroustrup,"The C++ Programming Language", Addison-Wesley., 1986.

[17] S. Chakravarthy et al., "HiPAC: A Research Project in Active, Time Constrained Database Management", in *TR XAIT-89-02*, Xerox Advanced Information Technology, Cambridge, MA, July 1989.

[18] S.B. Zdonik and D. Maier, "Object Oriented Fundamentals", in *Readings in Object-Oriented Database Systems*, 1990, 1-32.

A Appendix

```
class DPCDEF {

private :

   int  ChassisNum;
   int  UPmSlotNum;
   char DPCName[20];
   int  LogTMIBAddr;
   char SoftType;
   char SoftType2;
   char RedunGrpNam[25];
   int  CLCSetting;
   int  PrimeDLLServ;
   int  SecDLLServ;
   char ServClass;
   int  DesState;
   int  MaxNRTKBPS;
   int  CrKSPeriod;
   char Present;
   char Text[80];
   char PUMIB;

public :

   DPCDEF(void);
   ~DPCDEF(){ };
   void create( DPCDEF ScrDat, DPCDEFFlagType Flags, CACStatusType Result,
                RespLineType Response )
                                  /* Event Interface Modules */
   Set_dpcname(char *name);
   Set_dpc_upm_value(int upm_value);
}

/* Class Definitions of required Rules*/

class DPC_Name_uniq_Rule:public Rule{
 public :
   int  Condition(Reactive *obj, char *err_str, int argc, void **argv)
   {
        if ( (Cardinality(os_Set<DPCDEF*> &temp_set = DPC_set[: DPCName =
               (char*)argv[0] :]) > 0)
               return(1);
        return(0);
    }

   int  Action(Reactive *obj, char *err_str, int argc, void **argv)
   {
      sprintf(err_str,"DPCName %s already exists \n", (char *)argv[0]);
      return(1);
   }

   DPC_Name_uniq_Rule(char *in_name, Event *eventid, int in_enabled) : Rule(
```

```
            in_name, eventid, in_enabled) { };

    ~DPC_Name_uniq_Rule() { };

     static os_typespec* get_os_typespec();

};


class DPC_Upm_Range_Rule:public Rule{
 public :
    int  Condition(Reactive *obj, char *err_str, int argc, void **argv)
    {
          ( ((int)argv[0] == 1) || ((int)argv[0] == 12) ) ? return(1) : return(0);
    }

    int  Action(Reactive *obj, char *err_str, int argc, void **argv)
    {
       sprintf(err_str,"UPM value %d can only be 1 or 12 \n", (int)argv[0]);
    }

    DPC_Upm_Range_Rule(char *in_name, Event *eventid, int in_enabled) : Rule (
             in_name, eventid, in_enabled) { };

    ~DPC_Upm_Range_Rule();

     static os_typespec* get_os_typespec();

};

/* part of the code which creates rule instances along with the relevant events */

    Event *ev_dpc_name_type = new Primitive("DPC_Name_type_event");
    Event *ev_dpc_upm_type = new Primitive("DPC_Upm_range_type_event");

    Rule *dpc_name_rule = new(db) DPC_Name_uniq_Rule("DPC_Name_uniq",
                                          ev_dpc_name_type, 1); // 1 means enabled

    Rule *dpc_upm_range_rule = new(db) DPC_Upm_Range_Rule("DPC_Upm_Range",
                                    ev_dpc_upm_type, 1); // 1 means enabled
    char err_str[80];

DPCDEF::DPCDEF( ) {
        this->subscribe(dpc_name_rule);
        this->subscribe(dpc_upm_range_rule);
}

DPCDEF::Set_dpcname(char *name)
{
    if(Notify(this, "DPC_Name_type_event", err_str, 1, name)
       strcpy(DPCName, name);
}

DPCDEF::Set_dpc_upm_value(int upm_value)
```

```
{
    if(Notify(this, "DPC_Upm_range_tyep_event", err_str, 1, upm_value)
        UPmSlotNum = upm_value;
}

void
DPCDEF::create( DPCDEF ScrDat, DPCDEFFlagType Flags, CACStatusType Result,
        RespLineType Response )
{
  int   Chasinum, CLCset;
  char  present;
  RespLinType  DumResp;
  CACStatusType DumResult;

  do_transaction() {
                                        /* Range Check on UPmSlotNum */
  /*  if(( ScrDat.UPmSlotNum == 1) || (ScrDat.UPmSlotNum == 12)) {

      Result = cacError;
      strncpy(Response, "Error - slot numbers out of range";
      highlight(&Flags.UPmSlotNum);
    }
  */
  /* Note : Above is  the previous code which would be replaced by */

  if(!Notify((this, "DPC_Upm_range_tyep_event", err_str, 1, upm_value))
      abort();
  else {

    ScrDat.PUMIB = MakePUMIB(ScrDat.NetNum, ScrDat.ChassisNum,
                             ScrDat.UPmSlotNum, FALSE);
    *this(ScrDat);   /* Constructs the current object */

       /* Make sure ChassisNum and UPmSlotNum are unique - Uniqueness check */

/* similarly other rules will defined and made use of to check such conditions
in the remaining code which is not given here */

    Response = ErrDPCAlreadyExists;
    ..................
    ..............
    ..................
}
```