DESIGN AND IMPLEMENTATION OF MODELING AND VALIDATION FACILITIES FOR BUSINESS OBJECT DOCUMENTS

Haifei Li
Stanley Y.W. Su

Database Systems Research and Development Center
{hli, su}@cise.ufl.edu

As a real or virtual enterprise develops or makes use of more and more software application systems to manage all kinds of resources to conduct its business, the integration of these heterogeneous systems is becoming increasingly important. In a general integration scenario, application systems communicate with one another by sending data in certain data formats such as BODs (Business Object Documents) proposed by the Open Application Group (OAG). In order to ensure that meaningful communications take place, the data must satisfy the constraints of the source and target application systems.

In this report, we use the ECAA (Event-Condition-Action-Alternative action) rule paradigm to model the constraints of BODs at build-time and validate these constraints at run-time. The BOD modeling and validation system we developed consists of a set of build-time components and a set of run-time components. At build-time, a graphical user interface tool called XGTOOLS is used to model the "data types" (entities in the OAG terminology. Examples of data types are ITEMLOCATN and ITEMVALUE) of BODs as entity classes that contain attributes, constraints, events, rules and triggers. The modeled entity classes are translated into EXPRESS+ scripts for readability and dictionary interchange files for metadata transfer to other machines. EXPRESS+ is an extension of the EXPRESS modeling language. In addition to the language elements of EXPRESS, EXPRESS+ provides the language constructs for method specification and knowledge specification. Dictionary interchange files containing the metadata of BODs are transferred to a Metadata Manager. The Metadata Manager stores the metadata persistently and provides the application program interfaces (APIs) for accessing and modifying the persistent store. At build-time, the meta information of the entity classes of BODs is translated into Java source code and the corresponding ECAA rules are also translated into Java classes. These rules are enforced at the run-time when an application system generates a new BOD to be sent to other application systems or when an application system receives a BOD from another application system.

Since our approach to BOD modeling and validation is not tied to a particular business application, it can be used in any business application integration scenario. Furthermore, our modeling and validation techniques can also be used for modeling and enforcing constraints of objects other than BODs since the object model used in this work is generic.

This work is supported by the Advanced Technology Program of the National Institute of Standards and Technology.

SECTION 1
INTRODUCTION

In today's worldwide economy, many enterprises often need to team up to form a "Virtual Enterprise" [HAR97] in order to make the best use of people, data, hardware, software, and other resources. The integration [CHU96] of dissimilar computing systems and application software for the purpose of information sharing and software reuse are the key to success. However, the integration presents a difficult problem due to the fact that these participating enterprises generally have different data management systems, application software, operating systems and hardware systems. Integration of these heterogeneous systems requires a way of sending data and specifying operations among them. It will certainly simplify the task if a standard is used for sending data among systems and for specifying the operations that they want some other systems to perform.

The Open Application Group (OAG) was formed by a group of enterprise application software developers [OPE95] in February 1995. It proposed a standard business data interchange format for sending business data and operation specifications. The standard format is called Business Object Document (BOD). A BOD is created by a source application system and can be transferred to one or more target application systems. It contains enough supporting data to enable the destination applications to perform the operation specified in the BOD. As proposed by OAG, all data fields in a BOD are of type character string. This means that a source application will convert its data from their native data types (integers, reals, etc.) into character strings and pack these strings in the format specified by OAG. The target application systems will receive the physical character strings, unpack them and convert the individual character strings back into the target application's native data types for use by the target application systems.

In transferring data among heterogeneous systems, it is important that data be validated before a data transfer to make sure that the data satisfies the constraints as specified by OAG as well as the constraints of the source application. For example, a specific BOD as proposed by OAG may require that some selected data fields are required or the maximal length of a data field is 40-character long. Although data stored in the source application system can be assured to have been validated by the source system, the source application system may have some additional constraints associated with the data which is to be sent to other systems. For example, it may require more restrictive control on the sensitive data to be sent to a specific target application system. Therefore, it still has to check the data values to make sure that they satisfy the constraints before they are converted into character format to form the physical BOD. Similarly, the data transferred to a target application system needs to be validated by the target system to satisfy its own constraints.

One approach to do data validation is to leave the responsibility to the source and target application systems. One problem is that these application systems are legacy systems. They were not written with the expectation to send or receive BODs because BOD interchanges are needed due to system integration. Therefore, it will require that these legacy systems be modified and extended, not only to send and receive BOD data

but also to do data validations: a difficult and costly task. Thus, data interchange and validation needs to be done outside of the legacy application systems to minimize the impact to them. Although, some legacy application systems have done their own data validations inside their individual systems, additional data constraints or somewhat different constraints may be needed due to the system integration. These additional and/or modified constraints should ideally be checked and enforced outside of the legacy systems.

The approach presented in this report includes the following steps. First, we model BODs based on the views of both source and target systems using a GUI tool. As we pointed out before, the source and target views of a BOD can be different due to the different constraints seen by them. If this is the case, two conceptual models (one on the source side and the other on the target side) of the same BOD will be produced in this step. In the conceptual model of a BOD, data entities are modeled by a set of inter-related entity classes. All data attributes of these classes are modeled in the data types as used in the application systems (i.e., in their native data types). We capture the frequently used constraints of a BOD by keywords (e.g., Optional/Required Fields, Key Fields, Maximal Length, etc.) and less frequently used constraints by ECAA rules. ECAA rules can be used to specify more complex constraints such as inter-attribute constraints (e.g., the value of attribute A has to be equal to the sum of the values of attributes B and C) and inter-entity constraints (e.g., an instance of an entity type having a certain data property has to be associated with an instance of another entity type having some other data property). The provision of keyword constraints and ECAA rules in object modeling represents an extension to the traditional object model in which only attributes and methods are defined in an object class. Second, the keyword constraints are translated into ECAA rules. These and other ECAA rules that capture more complex constraints together with the entity classes that model the data of a BOD form the conceptual model of the BOD. This meta information are stored and managed by a metadata manager. Third, the conceptual models of all the modeled BODs are then translated into Java classes by a code generator. Some of these generated Java classes correspond to the entity classes and others represent the ECAA rules. "Get" and "Set" methods for all the attributes of entity classes are automatically generated for their corresponding Java classes. In these rule classes, methods are automatically generated based on the semantics of the CAA parts of the rules. These methods are executed when the events associated with the rules have been raised. Based on the conceptual model of a BOD, another code generator is used to generate the skeletal program code in Java for creating the instances of the BOD using the Set methods. The skeletal program code is to be used by the source application to create the BOD by providing the data values for the BOD. Another skeletal program code in Java is generated for the target application system to access the data in the BOD by using the Get methods. The main reason for using Java as the implementation language for the generated code is because it is platform-independent. The generated code can be considered as a part of the adapter, through which an application system is connected to a communication infrastructure such as CORBA, DCOM, or Java's RMI.

At run time, a source application creates a BOD by using the source skeletal program to "set" the values for the instances of the Java classes generated for the BOD. Rule codes are triggered to enforce the constraints of the source application. The

validated instances are then translated into a physical BOD (all data items are represented by character strings). The physical BOD is transferred to one or more target systems through the communication infrastructure. On the target side, a converter translates the physical BOD into the instances of Java classes, which represent the conceptual model of the BOD as seen by the target application system. All the data fields in the physical BOD are converted into the target system's native data types. The converted data values are moved into the instances of the Java classes generated for the target conceptual model. Rules associated with the target conceptual model are triggered to validate the data. The validated data is then accessed by the generated target skeletal program and moved into the target application system using its application program interface (API).

The proposed BOD modeling and validation approach allows the data provided by the source application system to be validated outside of the application system and before its transmission to some target system(s). Thus, data to be transmitted will satisfy the data constraints of OAG's specifications of BODs as well as those of the source system. This will avoid the transmission of nonsensical data created by, for example, programming errors. On the target side, data are validated to ensure that data constraints of the target system are satisfied before moving them into the target system.

In this work, ECAA rules are used to capture a variety of constraints associated with BODs; not only those specified by OAG but also those defined by application systems are supported. The concept and use of ECAA rules for constraint specification and enforcement is adapted from some existing active database management systems [ACT96, HAN93] such as HiPAC [DAY88, BUC95], Ariel [HAN92], Alert [SCH91], Sentinel [CHA94a], POSTGRESS [STO91], ODE [GEH91, GEH96], Starburst [LOH91, WID90] and our own work on OSAM*.KBMS [SU91, SU92, SU93, SU95, SU96a, SU96b, SHY96]. In active systems, database operations or user-defined operations can be treated as events, which can trigger the execution of rules. Each rule has a condition part, which evaluates to True or False. If the result of a condition evaluates to True, the action part of the rule is performed. Otherwise, the alternative action is performed. Different active systems provide different degrees of complexity that are allowed in the event and rule specifications. For example, some systems support composite and temporal events in addition to simple events. In the condition part of a rule, some systems allow the use of a query to retrieve data from a persistent store to check some data conditions. The action and alternative action parts of a rule may contain assignment statements, method calls, database operations, and/or general-purpose programming language statements. In this work, the operations of "setting" or "getting" data values are treated as events that may trigger the execution of rules (i.e., the CAA parts of rules).

In the remainder of this report, Section 2 presents the build-time and run-time architectures and their software components. Section 3 explains the build-time BOD modeling process in detail and with examples. Section 4 describes the run-time BOD validation process with examples. Section 5 gives a summary of this work, discusses the features and limitations of the work and identifies directions for future work.

SECTION 2
SOFTWARE ARCHITECTURE

We use an integrated manufacturing application environment as an example to show the application of the BOD modeling and validation tools we have developed. We would like to emphasize that the tools described in this work are applicable to any other integrated heterogeneous information system, which uses OAG's BODs as means of data transmission and operation specification. In an integrated manufacturing system, a number of heterogeneous application systems such as Manufacturing Execution System (MES), Enterprise Resource Planning (ERP), Capacity Analysis (CA), and Manufacturing Scheduling/Finite Scheduling (MS/FS) are connected to a common communication infrastructure through a number of adapters as illustrated in Figure 2.1. The communication infrastructure can be based on OMG's CORBA [OBJ92a, OBJ92b], Microsoft's DCOM [EDD98], Java's RMI [DOW98], etc. The work described in this report is independent of the communication infrastructure used to connect the distributed heterogeneous systems. The BOD modeling and validation tools we have developed can be considered as a build-time software system used by users of different application systems. It provides build-time tools for modeling BODs. Based on the conceptual models of BODs, Java codes that implement the semantics of BODs, BODs converters, and skeletal programs are generated. This generated code together with BOD converters and skeletal programs constitute a part of the adapters, which perform the run-time data validation and conversion of BODs. In the following two sections, we shall describe the build-time and run-time architectures and their software components.

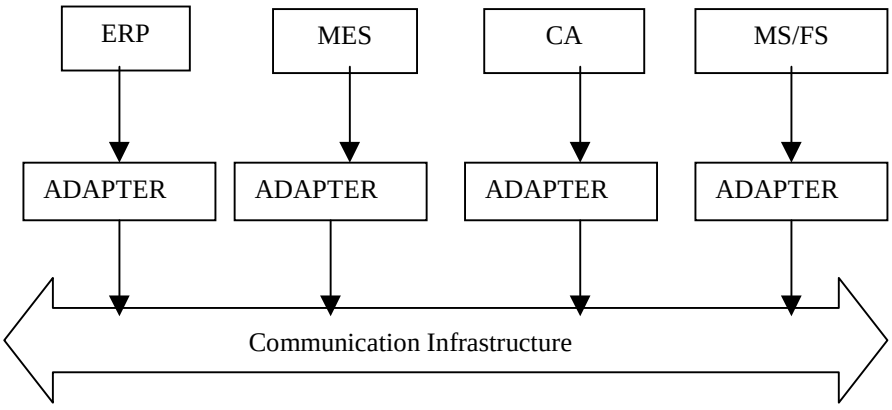

Figure 2.1 Enterprise Integration Architecture

2.1 Build-time Architecture and Software Components

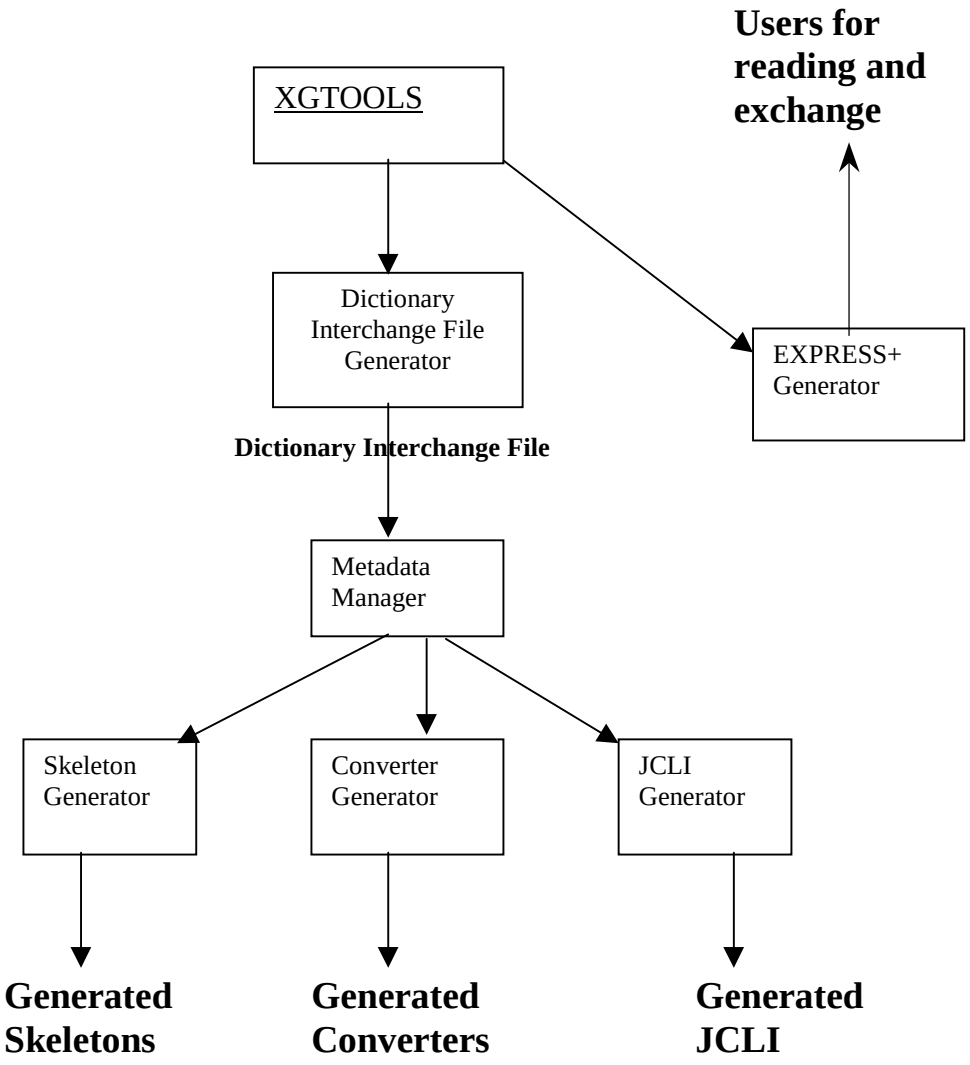

Figure 2.2 Build-time Architecture

Figure 2.2 shows the components of the build-time architecture. XGTOOLS [LAM92] is a set of general-purpose graphical user interface tools developed for modeling data, software components and processes as object classes. This tool set was developed for object and process modeling and is adopted for BOD modeling purposes. Each class is defined graphically by a set of attributes with keyword constraints, a set of methods, a set of associations with other object classes, and a set of ECAA rules. Thus, the underlying object model of the GUI is an enriched object model, which allows constraints, association types and knowledge rules to be specified. The tool set consists of

a graphic editor, a graphic browser, and a graphic querying tool. These tools are written in C and run on an AIX machine. As we mentioned before, each BOD is modeled as a set of inter-related entity classes with constraints and rules. Rules associated with each class can be entered into the GUI by using the simple rule editor within the graphic editor. The metadata associated with each modeled BOD are stored in an internal dictionary.

A generator called EXPRESS+ Generator has been developed to take the contents of the internal metadata dictionary and translate them into a textual language for easy readability. This language is called EXRESS+ since the syntax is patterned after the ISO's standard information modeling language EXPRESS [INT94]. In the present implementation, EXPRESS+ borrows most of its constructs from EXPRESS for modeling BODs. Its has the additional constructs for defining methods, and ECAA rules.

In Figure 2.2, another generator called Dictionary File Generator is used to take the metadata of the internal dictionary and generates a dictionary interchange file. The file is then transferred to and read by the Metadata Manager, which is written in Java and runs on an NT machine. The persistence of the metadata is supported by the serialization feature of Java. We note here that, if a GUI other than XGTOOLS (e.g., a commercial modeling tool) is used, the metadata captured by that tool can be put in the dictionary interchange form and be read by the Metadata Manager. Thus, multiple GUIs for BOD modeling can be accommodated.

The Metadata Manager provides the APIs for three other generators to access the metadata of BODs. JCLI Generator is responsible for generating a set of Java classes, some of which correspond to the entity classes of the BOD and others implement the rules associated with the BOD. We note here that the set of Java classes on the source side can be different from that of the target side since the source and target application systems' views of the same BOD can be different (e.g., different data constraints). Furthermore, the data value of the source can be different from that of the target (e.g. the values X, Y and Z of the source application system may be the same as the value 1,2 and 3 in the target application system). Thus, some data will have to be converted when moved from the source system to the target system. This is a data mediation task, which is out of the scope of this work but can be easily accommodated in the BOD translation, validation and transmission process.

The Skeleton Generator is responsible for generating a skeletal program for the source application system to establish the instances of the generated Java classes using Set methods, and a skeletal program for the target application system to get the data out of the instances of the Java classes using Get methods. The Converter Generator generates two converters based on the metadata of each BOD. One converter converts the populated Java classes' instances into a physical BOD on the source side and the other converters converts the physical BOD into the Java classes' instances on the target side.

2.2 Run-time Architecture and Software Components

Figure 2.3 shows the components involved in the run-time architecture. The source application modifies the generated source skeletal program by filling in the data values (in their native data types) that are to be used to build a BOD. This modified program runs in the same machine as the source application system. In this program, Set methods generated for the Java classes that implement the entity classes of the BOD are used to "set" the values of their attributes. Some of these Set methods will raise events to

trigger the execution of rules if there are constraints associated with the attributes they set. Exception conditions will be raised if the attribute values violate some constraints. If the BOD passes the validation phase, it is converted by the generated converter into a physical BOD. The physical BOD is then transmitted to one or more target application systems through the communication infrastructure. Although only one target application system is shown in Figure 2.3, multiple systems can receive the same BOD. In the adapter associated with a target system, the generated target converter converts the physical BOD into instances of the Java classes generated based on the conceptual model of the target system. Set methods are used to set the converted data (in native data types of the target system) to their corresponding attributes. At this time, constraint rules associated with some Set methods are triggered to enforce the constraints of the target system. The validated BOD is then accessed by the modified skeletal program to get the attribute values from the Java class instances. These values are then moved into the target application system using the APIs provided by it.

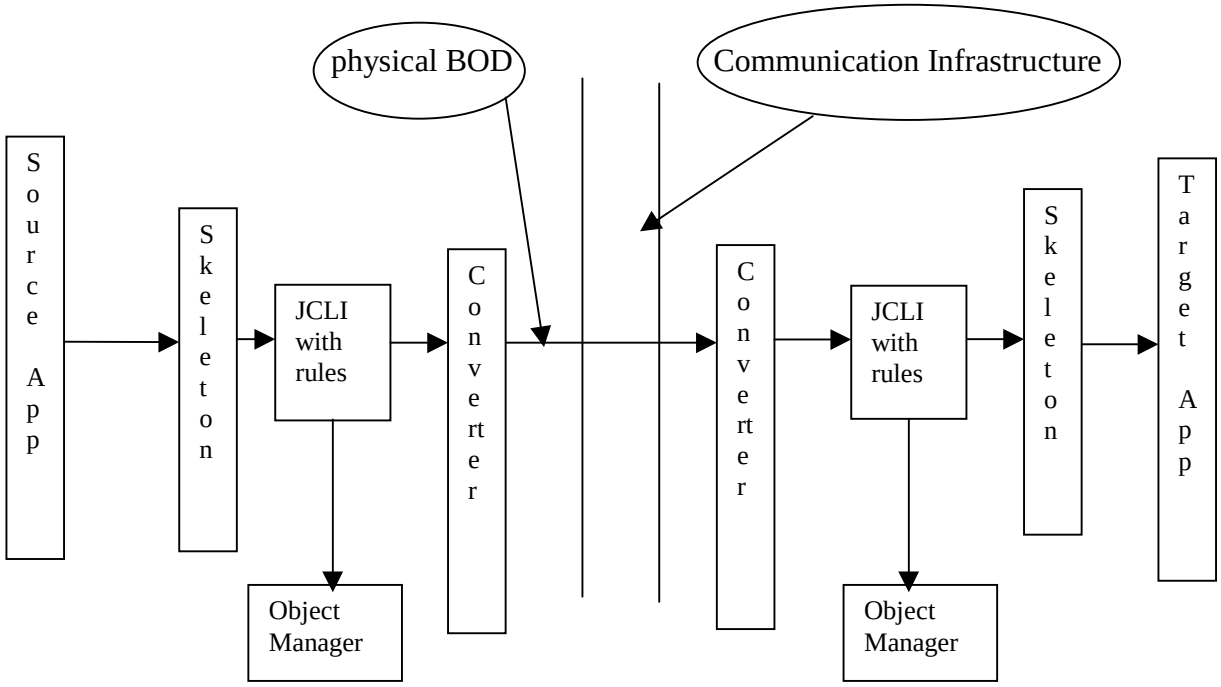

Figure 2.3 Run-time Architecture

In Figure 2.3, the Java classes that represent the source and target views of a BOD are called Java Call Level Interface (JCLI). They contain classes that implement rules. The validation of some of the rules requires the service of an object manager, which stores and maintains the instances of Java object classes during the run-time. For

example, the uniqueness constraint associated with an attribute requires the checking of all the instances of a class to make sure that a key attribute’s value is unique. Object manager is also used by the query processor, which is not a part of the report and is not shown here.

SECTION 3
MODELING OF BUSINESS OBJECT DOCUMENTS

This section describes in detail the build-time facilities of the BOD modeling and validation tool suite. The tool suite includes XGTOOLS, a JCLI generator (with ECAA rules), generator for converters (source and target) and generator for skeletal programs (source and target).

3.1 Introduction to OAG's Business Object Documents

3.1.1 Overview

The Open Applications Group, Inc. is a nonprofit consortium of enterprise application software developers, formed in February 1995 to create common standards for the integration of enterprise business applications. Member companies are building standard business service specifications. The scope of the work includes:

- Integration from enterprise planning and managing to extra-enterprise systems,
- Integration between enterprise planning and managing systems,
- Integration from enterprise planning and managing to enterprise execution systems.

OAG only defines specifications for business object interoperability between enterprise business applications. It does not build software and middleware transport mechanisms. At this time, OAG does not require applications to be certified for compliance. Each corporate member of the Open Application Group is committed to implement the Open Applications Group standard. If compliance becomes an issue in the future, the Open Applications Group will move to address the issue.

OAG calls the business objects communicated among business applications Business Object Documents (BOD). The definition of BOD and related materials are included in the Open Applications Group Integration Specification (OAGIS). The following subsection describes the BOD in OAGIS Release 5.

3.1.2 Business Object Document (BOD)

The Business Object Document is used to communicate a request and data from the originating business application to the destination business applications. Each Business Object Document includes supporting details to enable the destination business applications to accomplish the action.

BOD is a simple and flexible data model. It is similar to OEM (Object Exchange Model) [PAP95] used in TSIMMIS [GAR95] project which also addresses object exchange issue across heterogeneous information sources. Physically, objects of both models are character strings. However, there are some differences between the two models. First, OEM is a self-describing model, but BOD is not. Therefore, processing of a BOD needs extra work. In OEM, the labels given to the object are relative to the data source, but BOD defines the ontology shared by all information sources. The reason behind the difference is that BOD is limited to the domain of business application, but OEM is defined to be a general-purpose model. OEM has the corresponding query language called OEM-QL (Object Exchange Model-Query Language) that is similar to

SQL, but BOD does not define or adopt any underlying query language. The manipulation of BOD depends on the application vendors that implement the BOD standards. OEM is a free form data model in the sense that the meaning of fields of the object is independent of the position, just like the case of most specification and programming languages. However, BOD is position dependent. Each field has the fixed position in the BOD. Furthermore, delimiters such as blank space, tab character (\t) and line delimiter (\n) are not allowed in the BOD. The physical BOD is a long string without any physical separator, making it very hard to read by people.

A BOD is logically divided into several parts by five-character markers. For example, An MBBOD/MEBOD pair is used to mark the beginning and the end of a physical BOD. An MBDDA/MEDDA pair is used to mark the beginning and the end of a Data Definition Area. The very basic data element of a BOD is called a field. Each field has a Field Identifier and OAG has the authoritative definition for it. Examples are ITEM and BOM (Bill Of Material). Segments are used to pass complex values that require several characteristics or properties to be interpreted by the receiver. AMOUNT, CNTROLAREA, DATETIME, OPERAMT, QUANTITY, SENDER are the segments defined by OAG. Fields and segments are group into a larger unit called data type. One DDA has one data type and one or many occurrences.

Structurally, a Business Object Document consists of two areas:

- Control Area
- Business Data Area

The Control Area consists of a BSR (Business Service Request), Sender and DateTime. The Business Service Request consists of three items: VERB[10] (numbers within square brackets are the number of characters for attributes, not the reference number), NOUN[10], and REVISION[3]. The value within square brackets is the number of characters. The Sender is represented with the following structure definition: LOGICALID[10], COMPONENT[10], TASK[10], REFERENCEID[40], CONFIRMATION[1], LANGUAGE[2], CODEPAGE[15], AUTHID[50]. DATETIME is the segment used to specify the time and date of the creation of the Business Object Document.

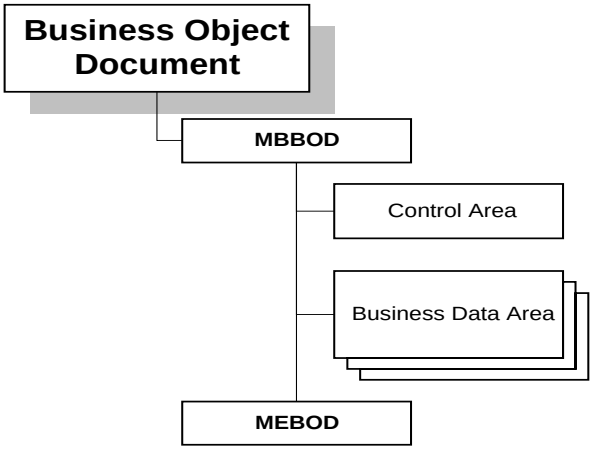

Figure 3.1 BOD Structure

The Business Data Area (BDA) of the Business Object Document contains all the codes, parameters and values needed to support the Business Service Request. For example, to post an inventory receipt, the Business Data Area will contain all the accounting information needed for the General Ledger application.

A BOD may contain one or many Business Data Areas supporting the same BSR. For example, a Business Service Request to post general ledger journal entries may contain one or many journal entries, each transferred in one Business Data Area of the same BOD.

The Business Data Area consists of two logical parts: **Data Definition Area and Occurrence of Values.** The following diagram depicts an example of the Business Data Area with various Data Definition Areas with different number of occurrences.

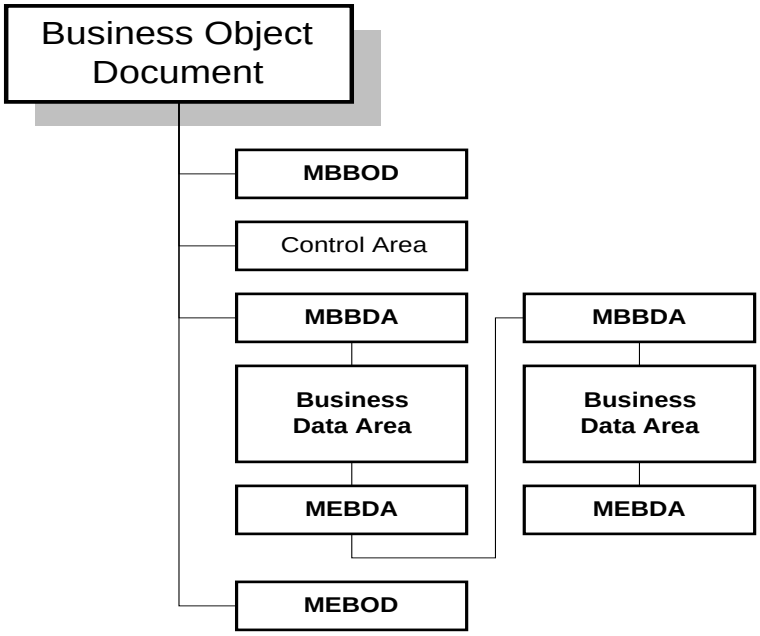

Figure 3.2 BOD That Has more than One BDA (Business Data Area)

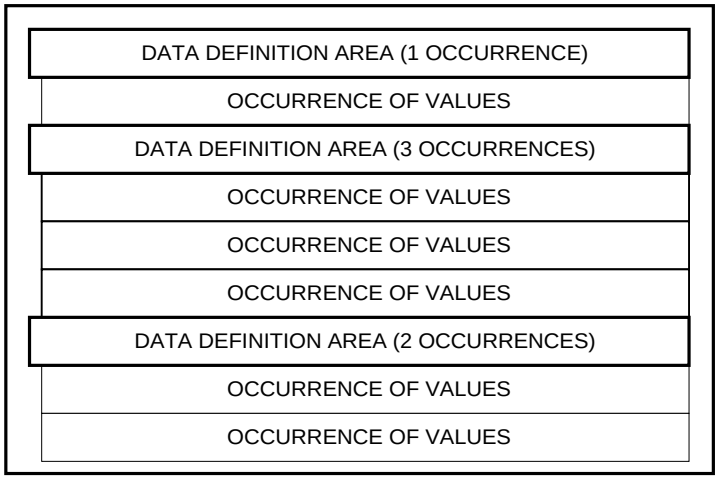

Figure 3.3 BDA (Business Data Area) Structure

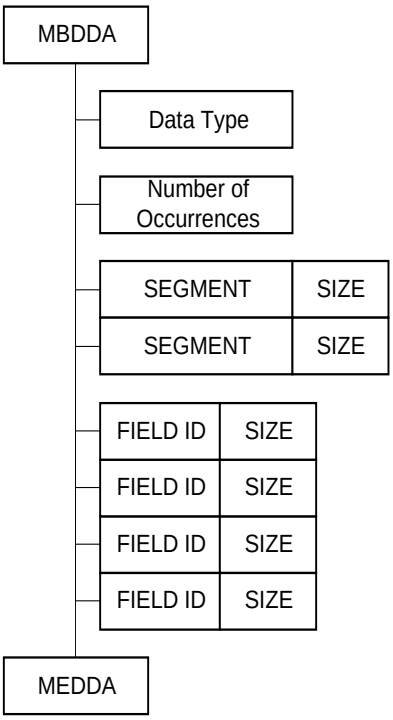

Figure 3.4 DDA (Data Definition Area) Structure

Each Data Definition Area consists of the following parts: Beginning Marker (“MBDDA”), Data Type Name, Number of Occurrences, Segment(s), Field Identifier(s), Ending Marker (“MEDDA”).

Other than the beginning and ending Markers, no other delimiters are used to separate any of the other parts of the Data Definition Area.

A Data Definition Area contains the definition of all segments and fields needed by a Data Type. Segments appear before fields. The following diagram is the DDA structure.

In order to give readers a concrete example of what a physical BOD looks like, a part of a physical BOD for SYNC_ITEM BOD is provided in Figure 3.5.

```
MBBODCNTROLAREA0194SYNC......ITEM......001ID........component.task.....
.1.....................................1ENISO150.......authid........
.....................................CREATION..19980515221937061 2+0000MB
BDAMBDDAITEMHEADER000001DATETIME..0033QUANTITY..0062QUANTITY..0062QUANT
ITY..0062QUANTITY..0062QUANTITY..0062ITEM......0004ITEMTYPE..0008UOM...
....0004BOMID.....0005BOMREVISON0010COMMODITY10010COMMODITY20010COMMODI
TY30010CONTRACTB.0009CONTRACTS.0009DESCRIPTN.0009DRAWING....0007GLENTIT
YS.0009GLNOMACCT.0009HAZRDMATL.0009ITEMCLASS.0009ITEMDEFN..0008ITEMRV..
..0006ITEMSTATUS0010LOTLEVEL1.0009LOTLEVEL2.0009LOTSNFLAG.0009NOTES....
.0005PARTNRID..0008PRODCTLINE0010PROPERTY1.0008UPC.......0003WARRANTY..
0008USERAREA..0086MEDDACREATION..19980515221937065 2+0000AVGRUNSIZE10...
.....................................1+EACH......LOTSIZEMAX10............
.........................1+EACH......LOTSIZEMIN10.....................
..................1+EACH......LOTSIZEMLT10.............................
........1+EACH......SHELFLIFE.10.....................................1
+EACH......itemitemtypeEACHbomidbomrevisoncommodity1commodity2commodity
3contractbcontractsdescriptndrawingglentitysglnomaccthazrdmatlitemclass
itemdefnitemrvitemstatuslotlevel1lotlevel2lotsnflagnotespartnridprodctl
inepropertyupcwarrantyMBUDAQUANTITY..0062MEUDATOTWEIGHT.10.............
.........................1+EACH......MBDDAITEMVALUE.
```

Figure 3.5 A Part of the SYNC_ITEM BOD

Figure 3.5 shows the CNTROLAREA and one BDA called ITEMHEADER. The dots (.) between characters are used to indicate blank spaces in the physical BOD. Contents from line1 to line 3 are CNTROLAREA. We can easily see that the verb is SYNC, noun is ITEM and revision is 001. Contents from line 3 to line 11 are the definition part of ITEMHEADER. Contents from line 11 to 20 are values for the corresponding segments or fields defined in the definition part. The last field in the definition part is USERAREA. The value for USERAREA appears in line 19 and line 20. The definition of USERAREA is between MBUDA (Mark Begin User Definition Area) and MEUDA (Mark End User Definition Area). The value for USERAREA follows MEUDA.

Figure 3.6 is the corresponding hierarchical structure for SYNC_ITEM BOD. Because every BOD must contain CONTROL AREA, it is not shown in the diagram. ITEMHEADER contains multiple BDA (Business Data Area) of ITEMVALUE and ITEMLOCATN. ITEMVALUE and ITEMVALUE also have their own underlying structures. What is shown in Figure 3.5 only includes CONTROL AREA and ITEMHEADER.

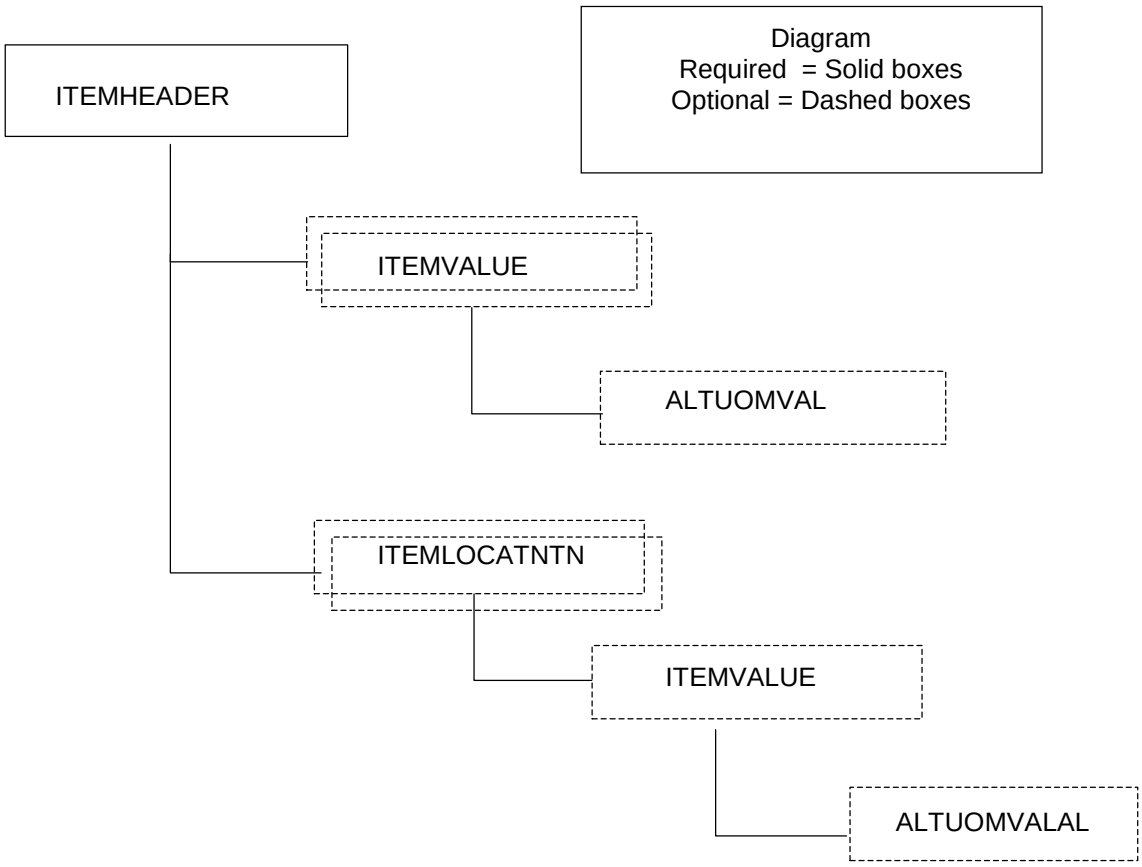

Figure 3.6 Hierarchical Structure for SYNC_ITEM

3.1.3 Extension to BOD Definition

OAGIS acknowledges that not all Field Identifiers necessary for every implementation can be predetermined. A special Field Identifier called **USERAREA** is the place where data unique to an implementation is defined. The **USERAREA** is always the last field identifier in the grouping of Field Identifiers in the Data Definition Area. These extensions do not change the definition of BOD because they are part of OAGIS. However, target application systems may not know how to process these extensions if they do not understand these special extensions.

3.2 Graphical User Interface for Modeling BODs

3.2.1 XGTOOLS Overview

XGTOOLS is a set of graphical tools developed for a knowledge base management system [SU95]. It has data modeling and process modeling facilities.

However, only the data modeling facility of this tool set is used in this work. The data modeling facility has three main functions: schema editing, schema browsing and graphical query. Schema editing is the visual editing tool for inserting, updating and deleting schemas, classes and attributes. Schema browsing is used to browse the existing schemas. Graphical query is used to inquiry the underlying KBMS if it is present. XGTOOLS is developed on RS/6000 and Sun workstations using Motif, C/C++. RS/6000 version of the tools is used.

3.2.2 BOD Modeling Using XGTOOLS

Below is the simple sequence of steps in using XGTOOLS:

1. Go to the directory in which XGTOOLS executables reside and begin a session by typing **xgtools** in a command window.
2. The InfoManager of XGTOOLS appears.
3. Select the Repository Directory. The default directory is the current working directory.
4. Select the working schema or type in the new schema you want to create.
5. Select the leftmost icon within the TOOLBOX by clicking on it.

The following is the results of an editing process. Figure 3.7 shows the graphical representation of the SYNC_ITEM BOD. In the figure, rectangles are entity classes, circles are selected attributes of these classes, and the bubble on the right side of each entity class, when clicked, will show all the other attributes of the class.

The following figures show the screen for entering constraints associated with attributes. Figure 3.8 and Figure 3.9 specify the RANGE constraint for an attribute. The minimum value is 1 and the maximum value is 999 in this case. Figure 3.10 specifies the MAXWIDTH constraint. The maximum width for this attribute is 10. Figure 3.11 shows the manually typed rule for LOTSNFLAG attribute of ITEMHEADER. It mandates that the value for LOTSNFLAG must be “L”, “S”, “N”, or “B”. The event is to be raised before the operation of setLOTSNFLAG. The rule is a static, active rule for verifying the value of LOTSNFLAG. The trigger links the event with the rule. In a more general case, an event can activate a structure of rules as specified by a trigger.

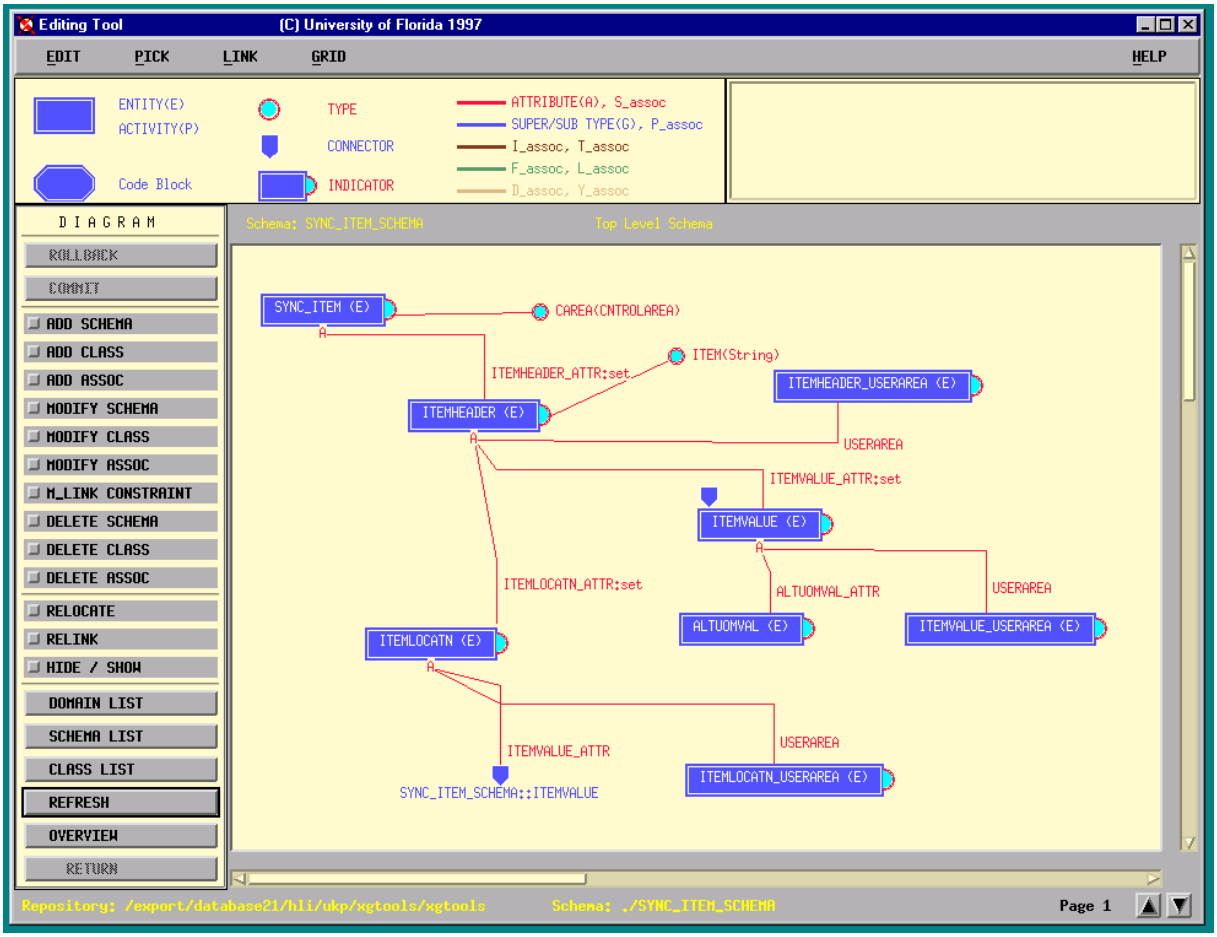

Figure 3.7 Graphical Representation for SYNC_ITEM BOD

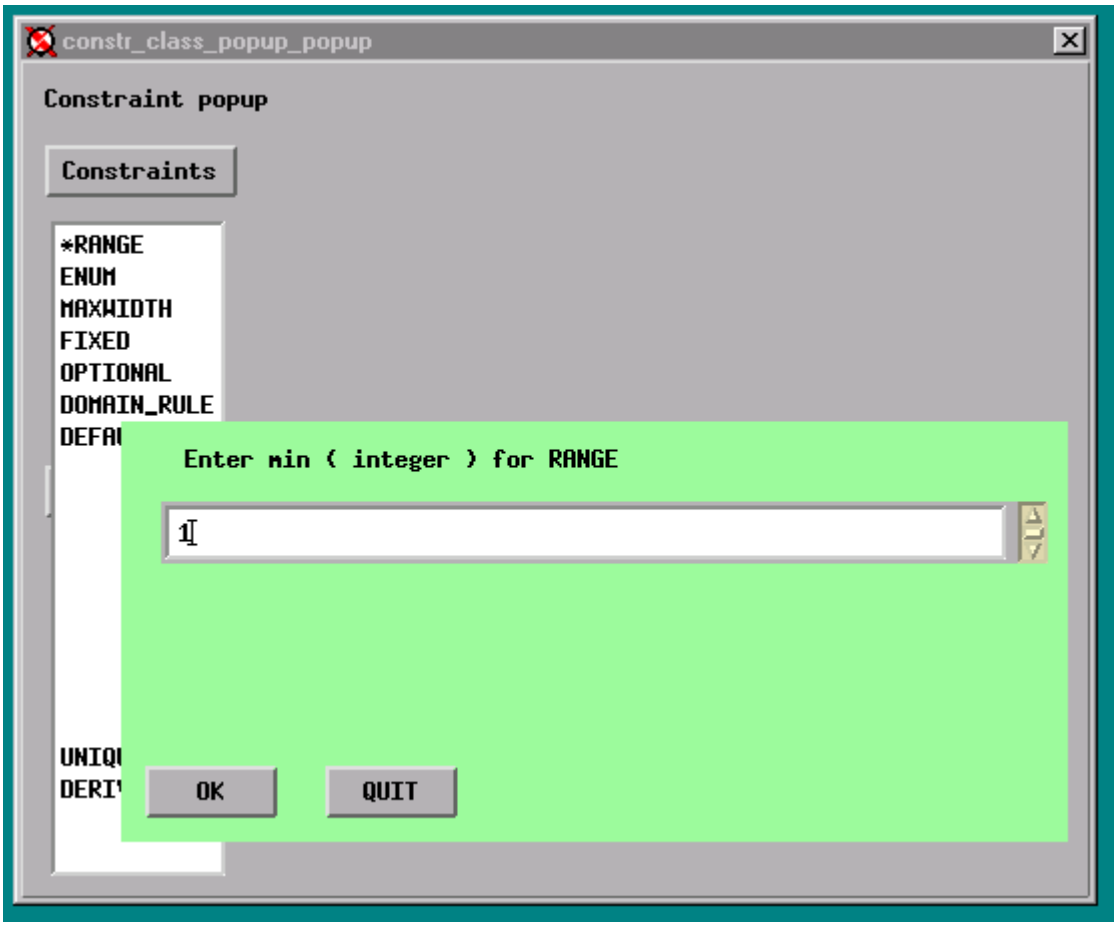

Figure 3.8 RANGE Constraint Specification Part 1

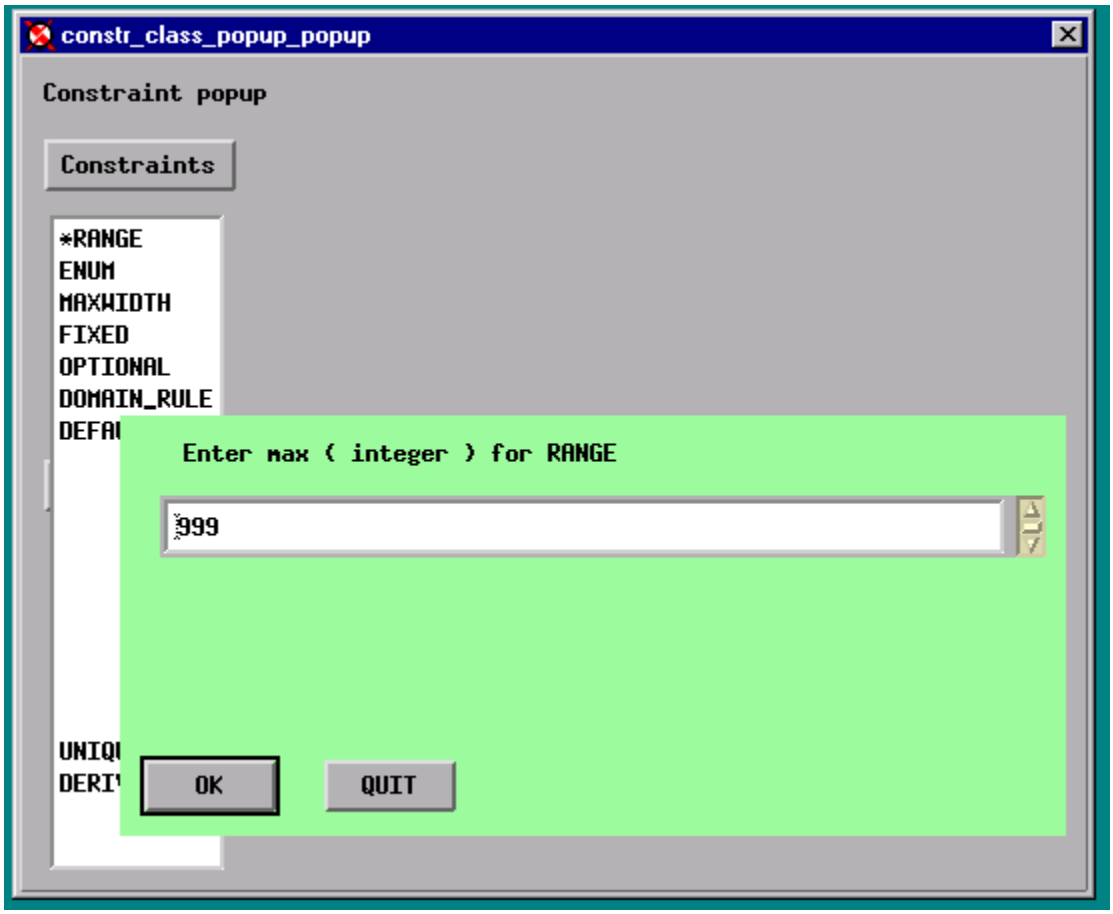

Figure 3.9 RANGE Constraint Specification Part 2

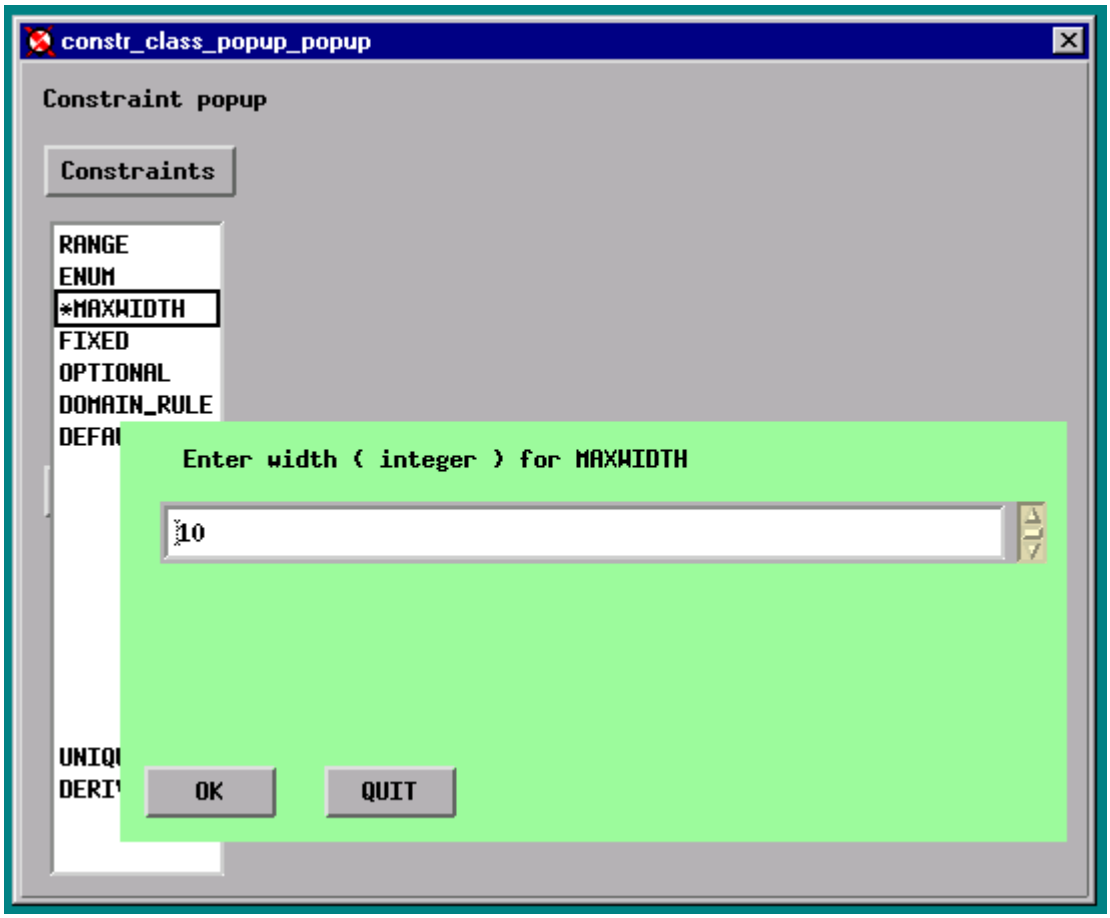

Figure 3.10 MAXWIDTH Constraint Specification

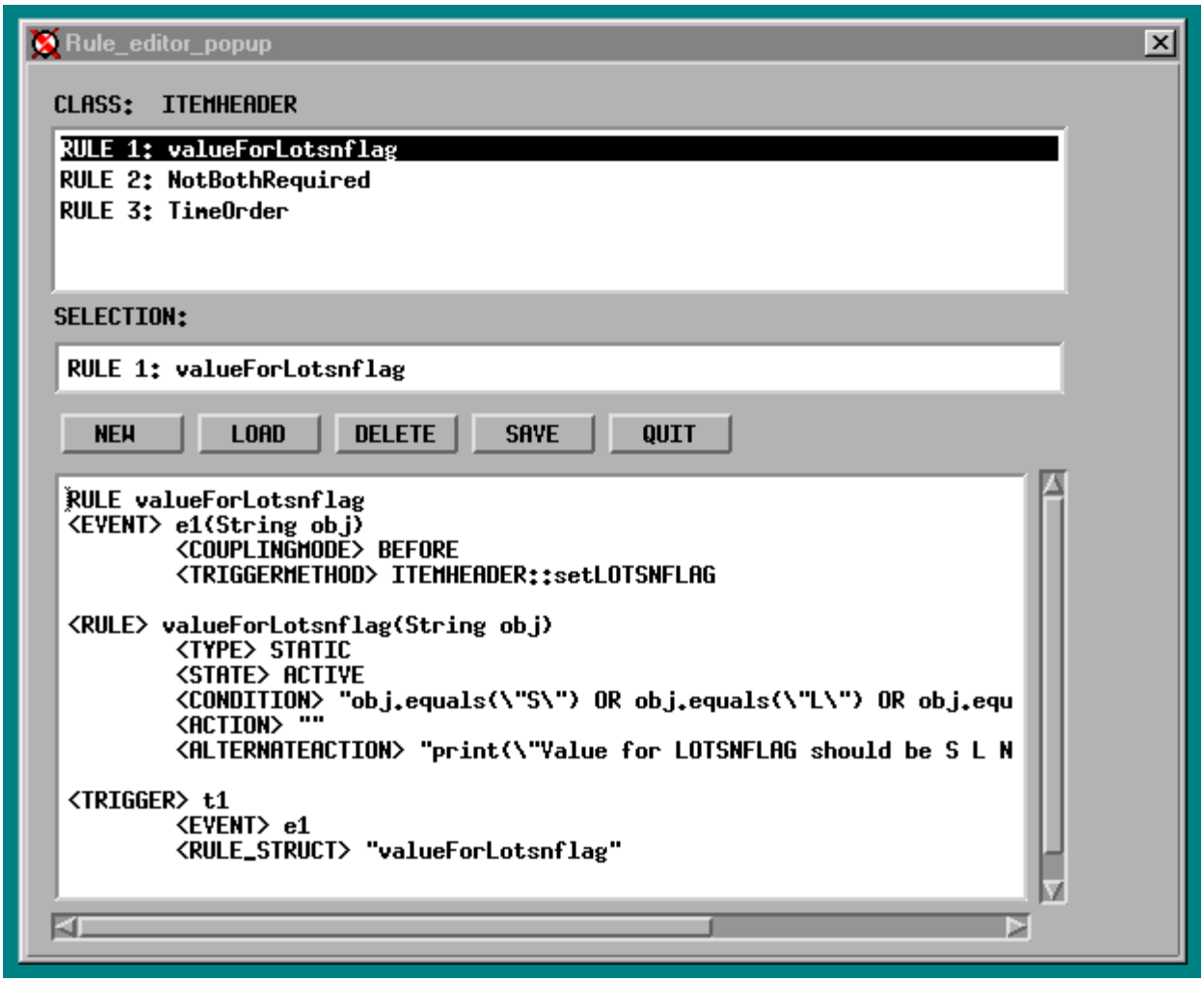

Figure 3.11 valueForLotsnflag Constraint Using Rule Editor of XGTOOLS

3.2.3 Naming Convention Used in the Modeling Process

Because of the large number of names used for attributes, data segments and data types, it is important to have a consistent naming convention for the system. Our convention for BOD modeling is detailed below.

Schema: It has the form: VERB_NOUN_SCHEMA. VERB and NOUN are the OAG-defined verb and noun in all upper case. SYNC and CONFIRM are examples of verbs, and ITEM and MO are examples of nouns.

Entity: All entity names are in the upper case, such as ITEMHEADER, and ITEMLOCATN. When an entity is used as an attribute, the attribute name is ENTITYNAME_ATTR. For example, ITEMHEADER_ATTR is an attribute name whose type is ITEMHEADER.

Field: use the name defined in the OAG definition. When fields are in the form name1-n, we have the following rules: If n <= 9, expand the name into name1, name2, name3, … If n > 9, regard it as a vector of string. For example, LOTLEVEL1-2 expands to LOTLEVEL1 and LOTLEVEL2. PROPERTY1-99 becomes a vector of string.

Segment: use the segment name followed by the qualifier. For example, in DATETIME_EFFECTIVE, the segment is DATETIME and the qualifier is EFFECTIVE. When segment name plus qualifier does not uniquely identify the attribute, additional fields of the segment are used. For example, in OPERAMT data segment, segment name plus qualifier plus type is used. The field name within a segment uses all upper case letters. For example, Year in DATETIME is YEAR.

USERAREA: The entity name is ENTITYNAME_USERAREA. ENTITYNAME is the entity that contains the USERAREA. For example, if POSUBLINE of ACKNOWLEDGE_PO BOD has one userarea, the name for it will be POSUBLINE_USERAREA. The attribute name that will be used in POSUBLINE is just USERAREA.

Functions: Set and get functions have the name: setNAME and getNAME. Example functions are setITEM and getLOTSNFLAG.

Generated files: In the following description, XXX means Verb, and YY means Noun. XXX_YYPopulate.java is the file containing the Java program for populating the Java classes using set functions. It is a source skeletal file. XXX_YYWritePBOD.java is the file containing the Java program for taking the top-level Java classes that represent a BOD, generating the physical BOD and putting it into a text file or directly sending it to the communication infrastructure. This is the source converter. XXX_YYReadPBOD.java reads the physical BOD, parses it and puts it into the Java classes. This is the target converter. XXX_YYRetrieve.java reads (gets) each field value from the Java classes, and prints it out. It is the target skeletal file, which is to be modified by the target system user to read data from the Java classes and move them into the application system using its APIs.

3.3 Translating Data Constraints into ECAA Rules

This section describes issues related to the translation of data constraints into ECAA rules. Rules may come from two different sources at build-time: keyword constraint specifications using the Schema Editor of XGTOOLS and manual addition of ECAA rules using a simple Rule Editor in the Schema Editor. Constraints captured by keywords are translated into ECAA rules. They are treated indiscriminately at run-time after they have been stored in the metadata repository.

One issue that needs to be addressed is the interaction between the event and the CAA part of the rule. Based on the problem to be solved, we take an efficient and simplified approach. Some active database systems have complicated subsystems to handle the event specification and detection, condition testing and rule execution model. For example, Sentinel has a fairly complete event specification language called SNOOP [CHA94b]. Events in SNOOP can be primitive types or composite events, and composite events are obtained by the application of event modifiers and event operators to primitive events. Accordingly, Sentinel has a component called event detector to capture and

analyze events. In Ariel, condition testing is done by a discrimination network composed of a special data structure for testing single-relation selection conditions efficiently, and A-TREAT algorithm for testing join conditions. HiPAC proposed a nested transaction model (NTM) for the rule execution. In our case, the execution of the triggering method is treated as the event, so the event specification and detection is relatively easy and efficient. For the BOD validation task, most of the triggering methods are set and get functions, and these functions can be automatically generated. Therefore, we have full control of both the triggering method and the rule in the implementation. Function calls are added to the rule class before and/or after the actual codes for a set/get function. Since the semantics of rules is implemented in the source code after the code generation, a separate component to capture event at run-time is not needed. Condition testing is bound to the Action and Alternate-action part. It is a part of the rule class which will be discussed later.

ECAA rules are treated as objects in this system. They are divided into two main types, namely, class-level rules and instance-level rules. A class-level rule is applicable to all the instances of a class whereas the instance-level rule is applicable to specific instances, possibly from different classes. While some systems like POSTGRES implement instance-level rules (POSTGRES is an extended relational system. An instance-level rule is a tuple-level rule), most systems implement only class-level rules. The reason behind the selection is efficiency and simplicity. While an instance-level rule may provide a finer granule of control of an object, the system has to monitor changes to every single object. The class-level rule is enough for BOD validation purposes. In this project, most rules are constraints on object attributes. Examples are RANGE and MAXWIDTH constraints. When a RANGE constraint is defined for an attribute of a class, it must apply to all the created objects of that class. It is not necessary to explicitly specify that each object follows the constraint.

Even if the decision is made to use class-level rules, there are still two alternatives that can be used in rule translation. First, a class can be created for each rule. The condition and action parts of the rule will be implemented as methods of the newly created class. At run-time, only one object of the class will be created. The second alternative is to create an abstract rule class and let rule categories such as UNIQUE, DEFAULT, and MAXWIDTH inherit from the rule class. Each rule is an object of its rule category. At run-time, one object will be created for one rule, and some data that are unique to the rule are passed to the object through the constructor of the class.

Although the first approach creates more classes than the second approach, it is selected for the following reasons. First, the semantics of rule deletion is not very clear for the second approach. If the rule to be deleted is the last rule in that rule category, should the object of the rule class be deleted as well as the class definition? Second, if a user manually defines a rule, the system must assign a unique rule category for that rule. The rule category management will be complicated if the number of manually defined rules is large.

Section 3.7 has a concrete example of translating the RANGE constraint into a Java class. At run-time, an instance of that rule class is created and the call to a fireRule function is made. The fireRule function implements the logic of condition, action and alternative-action parts of the rule.

3.4 Generation of EXPRESS+ Representation of BODs

EXPRESS+ scripts are generated by clicking the "EXP+" icon on the InfoManager of XGTOOLS. Internally, XGTOOLS will invoke a program to get the meta information of a BOD from the internal dictionary file of XGTOOLS and translate it into EXPRESS+ representation.

Figure 3.12 shows the output of the generator showing the EXPRESS+ representation of the SYNC_ITEM. ECAA rules are shown in the EXPRESS+ schema by a list of Events, Rules and Triggers specifications in the KnowedgeSpec section. We separate events, rules and triggers to allow more flexible specifications in relating a structure of events (composite events) with a structure of rules.

3.5 Generation of Dictionary Interchange File

Figure 3.13 is a part of the generated dictionary interchange file for the SYNC_ITEM BOD.

```
SCHEMA SYNC_ITEM_SCHEMA;

ENTITY ITEMVALUE_USERAREA;
      QUANTITY_ITEM : OPTIONAL QUANTITY;
END_ENTITY;


ENTITY ITEMHEADER_USERAREA;
      QUANTITY_TOTWEIGHT : OPTIONAL QUANTITY;
END_ENTITY;

ENTITY ITEMHEADER;
      ITEMVALUE_ATTR : OPTIONAL SET of ITEMVALUE;
      ITEMLOCATN_ATTR : OPTIONAL SET of ITEMLOCATN;
      ITEM : String(50);
      ITEMTYPE : OPTIONAL String;
      UOM : OPTIONAL String;
      BOMID : OPTIONAL String;
      BOMREVISON : OPTIONAL String;
      COMMODITY1 : OPTIONAL String;
      COMMODITY2 : OPTIONAL String;
      COMMODITY3 : OPTIONAL String;
      CONTRACTB : OPTIONAL String;
      CONTRACTS : OPTIONAL String;
      DATETIME_CREATION : OPTIONAL DATETIME;

END_ENTITY;

ENTITY ITEMVALUE;
      ALTUOMVAL_ATTR : OPTIONAL ALTUOMVAL;
      COSTTYPE : String;
      GLENTITYS : String;
      GLNOMACCT : String;
      OPERAMT_UNIT_F : OPERAMT;
      VALUECLASS : String;
      DATETIME_EFFECTIVE : OPTIONAL DATETIME;
      DATETIME_EXPIRATION : OPTIONAL DATETIME;
```

```
      DESCRIPTN : OPTIONAL String;
      NOTES : OPTIONAL String;
      OPERAMT_UNIT_T : OPTIONAL OPERAMT;
      USERAREA : OPTIONAL ITEMVALUE_USERAREA;

END_ENTITY;
ENTITY SYNC_ITEM;
      ITEMHEADER_ATTR : SET of ITEMHEADER;
      CAREA : CNTROLAREA;
END_ENTITY;

.
.
. /* other entity classes are not shown here */

END_SCHEMA;
```

Figure 3.12 EXPRESS+ Representation

```
METHODSPEC ms in SYNC_ITEM_SCHEMA

      QUANTITY ITEMVALUE_USERAREA::getQUANTITY_ITEM();
      Void ITEMVALUE_USERAREA::setQUANTITY_ITEM(QUANTITY val);

      String ITEMLOCATN_USERAREA::getPARTMAXGRP();
      Void ITEMLOCATN_USERAREA::setPARTMAXGRP(String val);

      String ITEMLOCATN_USERAREA::getUNITMAXGRP();
      Void ITEMLOCATN_USERAREA::setUNITMAXGRP(String val);

      String ITEMLOCATN_USERAREA::getBUYPLNCODE();
      Void ITEMLOCATN_USERAREA::setBUYPLNCODE(String val);

      String ITEMLOCATN_USERAREA::getQLTYFLAG();
      Void ITEMLOCATN_USERAREA::setQLTYFLAG(String val);


      CNTROLAREA SYNC_ITEM::getCAREA();
      Void SYNC_ITEM::setCAREA(CNTROLAREA val);

END_METHODSPEC;

KNOWLEDGESPEC RS in SYNC_ITEM_SCHEMA


<EVENT> e1(String obj)
      <COUPLINGMODE> BEFORE
      <TRIGGERMETHOD> ITEMHEADER::setLOTSNFLAG

<RULE> r1(String obj)
      <TYPE> STATIC
      <STATE> ACTIVE
```

```
      <CONDITION> "obj.equals(\"S\") OR obj.equals(\"L\") OR
obj.equals(\"B\") OR obj.equals(\"N\")"
      <ACTION> ""
      <ALTERNATEACTION> "print(\"Value for LOTSNFLAG is wrong!\")"

<TRIGGER> t1
      <EVENT> e1
      <RULE_STRUCT> "r1"


END_KNOWLEDGESPEC;
```

Figure 3.12 –Continued

In this example, ITEMVALUE, an entity of SYNC_ITEM, is defined. It contains attributes like COSTTYPE, GLENTITYS, VALUECLASS, GLNOMACCT, and DESCRIPTN. Each attribute has some fields to describe it. For example, DESCRIPTN is an optional attribute. It is of String PrimitiveType. The signatures for the set and get functions of each attribute of the entity are automatically generated. In this example, the set and get functions for attribute COSTTYPE and GLENTITYS are shown. The EVENT, RULE and TRIGGER parts are similar to what has been described in EXPRESS+. The rule in this example means that the field DESCRIPTN should be less than 80 characters. That is to say, it enforces the MAXWIDTH constraint.

```
      <CLASS> ITEMVALUE ENTITY
            <ATTRIBUTE> COSTTYPE REQUIRED PrimitiveType     String
            <ATTRIBUTE> GLENTITYS REQUIRED PrimitiveType    String
            <ATTRIBUTE> VALUECLASS  REQUIRED PrimitiveType  String
            <ATTRIBUTE> GLNOMACCT   REQUIRED PrimitiveType  String
            <ATTRIBUTE> DESCRIPTN   OPTIONAL PrimitiveType  String
            <ATTRIBUTE> NOTES OPTIONAL    PrimitiveType     String
            <ATTRIBUTE> ALTUOMVAL_ATTR OPTIONAL DataType ALTUOMVAL
            <ATTRIBUTE> OPERAMT_COST_F REQUIRED DataSegment OperAmt
            <ATTRIBUTE> USERAREA    OPTIONAL    USERAREA IV_USERAREA
            <ATTRIBUTE> DATETIME_EFFECTIVE OPTIONAL DataSegment
            DateTime

            <METHOD> String getCOSTTYPE()
            <METHOD> Void setCOSTTYPE(String obj)
            <METHOD> String getGLENTITYS()
            <METHOD> Void setGLENTITYS(String obj)
      .
      .  /* other metohds are not shown here */
      .

      <EVENT> EMAXWIDTH16(String obj)
            <COUPLINGMODE>  BEFORE
            <TRIGGERMETHOD>   Sender::setDESCRIPTN

      <RULE> RMAXWIDTH16(String obj)
            <TYPE> STATIC
            <STATE>  ACTIVE
            <CONDITION>  "strlen(obj) <= 80"
            <ACTION>  ""
```

```
        <ALTERNATEACTION>
        "print(\"Max string length is exceeded! \")"

<TRIGGER> TMAXWIDTH16
        <EVENT> EMAXWIDTH16
        <RULE_STRUCT>  "RMAXWIDTH16"
```

Figure 3.13 Dictionary Interchange File

3.6 Importing Metadata into the Metadata Repository

In order to facilitate the import process, a set of syntactic rules is defined for the dictionary files. JavaCC [SUN] is utilized to parse the dictionary files. When JavaCC confirms that there is no error in the dictionary files, it writes the data into a number of Java classes. Because these classes implement the serializable interface, the data will be persistently stored in the metadata repository when the program exits.

3.7 Generation of Java Call Level Interface and Rule Code

Conceptually, the metadata of each BOD forms the following hierarchical structure.

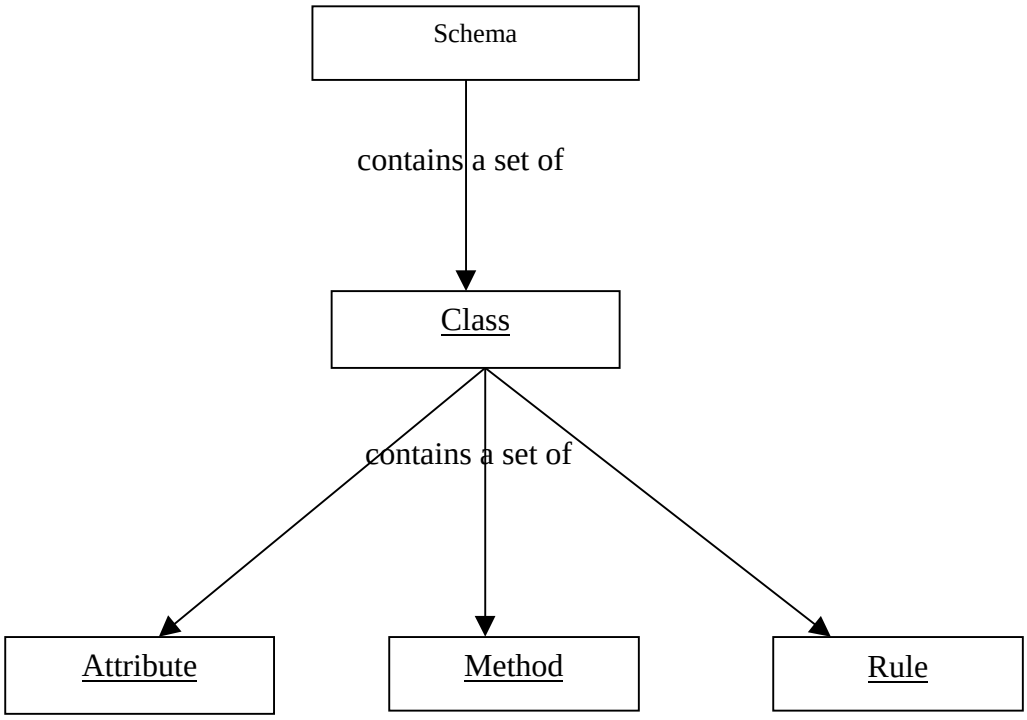

Figure 3.14 Hierarchical Structure for the Metadata

Each box in the diagram has the corresponding Java class in the metadata repository. The translation of the metadata into Java classes begins from the schema. At first, all the classes of the schema are fetched from the repository. Then they are translated into Java classes in the following way. Attributes of the class are translated into the corresponding private member data of the Java class. Methods of the class stored in the repository are the method signatures for set and get functions. They are translated into the corresponding public member functions of the Java class. The translation of rules needs a more detailed explanation. In addition to the generation of rule code for each rule, additional Java statements are inserted into the trigger method to enforce the rule.

Six classes are used for the translation task. They are MainGen, SchemaJavaCode, ClassJavaCode, AttributeJavaCode, MethodJavaCode and RuleJavaCode. The pseudocode descriptions of each class are given below:

MainGen: This is the top-level class for the translation.

- Create an object of SchemaJavaCode;
- Call the function SchemaCodeGen of SchemaJavaCode class.

SchemaJavaCode:

- Get all the classes of the schema from the Metadata Manager, put the result into a vector;
- For each element of the vector, create an object of ClassJavaCode;
- Call the function ClassCodeGen of ClassJavaCode class.

ClassJavaCode:

- Create a text file to hold the generated code;
- Generate the copyright and related information;
- Create an object of AttributeJavaCode, and call the function AttributeCodeGen of AttributeJavaCode class;
- Put the return value of AttributeCodeGen into a string called initcode which holds the initialization code for the vector data type;
- Create an object of RuleJavaCode and call the function RuleCodeGen;
- Put the return value of RuleCodeGen into a string called rulecode which holds the method call to the rule class;
- Output initcode after the code for a constructor;
- Create an object of MethodJavaCode and call the function MethodCodeGen;
- Output rulecode after initcode.

AttributeJavaCode:

- Get all attributes of the class, put the result into a vector;
- For each element of the vector, generate the corresponding member data according to its individual type.

MethodJavaCode:

- Get all the attributes of the class, put the result into a vector;
- For each attribute of the vector, do the following:
- Access the metadata repository to see if there is a before rule defined for the get/set function of the attribute. If so, create an object of that rule class and call the fireRule function;
- Generate the set or get function according to its individual type;
- Access the metadata repository to see if there is an after rule defined for the get/set function of the attribute. If so, create an object of that rule class and call the fireRule function.

RuleJavaCode:

- Get the events associated with the class from the Metadata Manager;
- Get the trigger associated with the events;
- Get the rules associated with the trigger;
- For each rule:
  - Get the Condition part, do some conversion, and put it into the CONDITION part of the if statement;
  - Get the Action part, do some conversion, and put it into the THEN part of the if statement;
  - Get the Alternate-action part, do some conversion, and put it into the ELSE part of the if statement.

In order to facilitate our discussion, we assume that we want to generate the code for the class CLASS1. CLASS1 has one integer attribute called ATTR1. There is a before rule RX, which is used to check the range of the attribute value for ATTR1 before the "set" function of ATTR1 is executed. If the attribute value does not fall into the range, an error message is printed.

In order to support QP (the Query Processor), which is closely related to our work here, class CLASS1 is defined as a subclass of QEObj which contains all the information necessary for query processing. For the same reason, CLASS1 implements a public function called updateAll, which will be called by the update statement of the query language. ATTR1 is defined as Integer class instead of the primitive type int because it is necessary to check whether the attribute is NULL or not. If the attribute was defined as the primitive type int, there will be no way to do the checking; at least in the Java language. Because a before-rule is defined for setATTR1 method of CLASS1, an object of the rule class RX is created and the fireRule function of the object is called before the actual assignment is done.

We now describe the Java code generated for the rule RX. In the example, RX class has a private attribute for ATTR1, a constructor, a public function fireRule and other constructs. The attribute holds the value that will be checked for its range. The constructor passes the parameter value to the attribute. The public function fireRule checks whether the value falls within the range of, say, 0 to 999. The function will print an error message if the value is out of the range.

It is necessary to make a couple of comments about the rule illustrated here. First, this rule does not have an action part, but has an alternative-action part. The alternative-action in the example only prints an error message. Ideally, it should call an exception management agent to handle the error. A future version will handle the exception handling of rules. Second, ECAA rules generated from keyword constraints are often simpler than those specified by modelers. The rule used here is from the RANGE keyword constraint. However, the simple example used here does not imply that it is not possible to generate more complicated rule class.

3.8 Generation of Java Skeletal Programs

Because of the complexity of the BOD structure, users may have difficulty with the manipulation of a BOD if a skeletal program is not provided for them to use. A skeletal program is one that allows the user to populate or retrieve data from a BOD. There are two skeletal programs: one is used by an application system on the source side to populate Java classes with native data, the other is used on the target side to retrieve

the data in their native data types from a transmitted BOD. Two converters are also generated for each BOD. A source converter takes the Java objects representing the BOD and converts them into a physical BOD in a character string format. The target converter takes the physical BOD and converts it into the Java object representation. When the programmer of the application vendor gets a skeletal program, he/she must revise it to meet his/her unique needs.

Skeletal program generation uses a recursive algorithm. The data segment of a BOD may have subsegments as its attribute values (Actually, only CNTROLAREA has DATETIME and SENDER as its subsegments. All other segments only contain the primitive attributes). Similarly, the data type of a BOD may have attributes defined over other data types. In generally, the level of nesting of attributes is not known until at run-time. The recursive nature of the BOD structure calls for the use of a recursive algorithm.

The algorithm for generating the source skeletal progam is similar to the one for generating the target skeletal program. However, there are some differences. On the source side, constant values for primitive data types are used to "set" the attributes of the entity classes. An application vendor programmer is supposed to replace these constant values with program variables. On the target side, a program variable is used to hold the result of a "get" method. After the value of the program variable is returned, the application can manipulate it in its own way.

3.8.1 Algorithm to Automatically Generate the Source Skeletal Program

- Generate statements to handle the CntrolArea of BOD;
- Call genOneDataType (the parameter is the top level BOD Data Type. For SYNC_ITEM, it is ITEMHEADER);
- Generate additional statements to do the post genOneDataType work.

```
genOneDataType(Attribute att)
/* att represents a BOD data type which may contain data segments, primitive fields, USERAREA
and sub data type */
begin
        get all the DataSegment of att;
        for each DataSegment
                generate the Set function for each individual field of the DataSegment;

        get all the PrimitiveType of att;
        for each PrimitiveType
                generate the Set function for each PrimitiveType;

        get the USERAREA ( an attribute) of att, if there is one
                generate the Set function for each attribute of the USERAREA

        /* In the physical layer, the subDataType is at the same level as that of DataType */
        get all the subDataType of att;
        for each sub-att
                call genOneDataType(sub-att)
end
```

3.8.2 Algorithm to Automatically Generate the Target Skeletal Program

The first task for the target skeletal program is to call the target converter to parse the physical string. This section only describes the algorithm for generating the target skeletal program (i.e., template for the use of "get" method of the JCLI). The algorithm for the target converter is described in a later subsection.

- Generate statements to handle the CntrolArea of BOD;
- Call genOneDataType (the parameter is the top level BOD Data Type. For SYNC_ITEM, it is ITEMHEADER);
- Generate additional statements to do post genOneDataType work.

```
genOneDataType(Attribute att)
/* att represents a BOD data type which may contain data segments, primitive fields, USERAREA
and sub data type */
begin

        get all the DataSegment of att;
        for each DataSegment
                generate the Get function for each individual field of the DataSegment;

        get all the PrimitiveType of att;
        for each PrimitiveType
                generate the Get function for each PrimitiveType;

        get the USERAREA (an attribute) of att, if there is one
                generate the Get function for each attribute of the USERAREA

        /* In the physical layer, the subDataType is at the same level as that of DataType */

        get all the subDataType of att;
        for each sub-att
                call genOneDataType(sub-att)

end
```

3.9 Generation of Source and Target Converters

3.9.1 Algorithm to Automatically Generate the Source Converter

This algorithm is similar to the algorithm for generating the source skeletal program. However, there are special features that need to be considered. Since the physical BOD is a character string, all values must be converted to string for output. For attributes with data segment, utility functions that pad the attribute with zero or blank spaces at the left or right side of the string must be provided. Because each attribute appears in the definition area and the value occurrence area, two accesses are necessary: one is to append the name to the definition list, the other is to append its value to the value occurrence area.

- Generate statements to produce the markers such as MBBOD, MBBDA;
- Generate statements to handle the CntrolArea of BOD;
- Call pBod (Attribute, level, String identifier); // identifier is the concatenate of type and level.

- Generate additional statements to do post pBod work.

```
pBod(Attribute att, int level, String identifier)
/* att represents a BOD data type which may contain data segments, primitive fields, USERAREA
and sub data type */
begin

        /* data segment appears before primitive attributes in the OAG definition */
        get all the DataSegment of att;
        for each DataSegment
                generate the character form for each individual attribute of the DataSegment;

        get all the PrimitiveType of att;
        for each PrimitiveType
                generate the character form for each PrimitiveType;

        get the USERAREA of att, if there is one
                generate the character form for each attribute of the USERAREA


        /* In the physical layer, the subDataType is at the same level as that of DataType */
        get all the subDataType of att;
        for each sub-att
                create value for ident
                call pBod(sub-att, level + 1, ident)

end
```

3.9.2 Algorithm to Automatically Generate the Target Converter

If the application receives a BOD and wants to incorporate its data into its domain, the physical BOD string needs to be parsed from its physical representation into an internal representation. Ideally, BODs should be self-describing, and can be parsed without any outside information. This allows a generic program to be written to parse any BOD string. Unfortunately, BODs, as currently specified by OAG, do not fully satisfy this requirement. It is not possible to know, for example, to which Data Definition Area (DDA) a particular field or another DDA belongs. The parent/child relationships within a BOD are specified in an ambiguous way.

As a result of the above limitation of the architecture of BODs as defined in the Open Application Group Integration Specifications (OAGIS), it is impossible to write a generic program that can parse BODs without apriori knowledge of these BODs.

In our work, we capture the metadata of all BODs in the Metadata Manager and develop a general converter generator to make use of the metadata and BOD strings to produce converters for different BODs. These converters take the physical BODs and convert them into instances of their corresponding Java classes. In the following algorithm, it is assumed that specific knowledge about the parent / child relationship of each BOD is known.

- Find “MBBOD” at the beginning and “MEBOD” at the end. Otherwise, raise an exception.

- Find the “CNTROLAREA” to the first character before “MBBDA”. This part should include all the correct control area data. Otherwise, raise an exception.
- Find the “MBBDA” and “MEBDA” pair (“MEBDA” is the string just before “MEBOD”). Otherwise, raise an exception.
- Following is the handling of DDAs and their occurrences of data values
- Loop:
  - Get the name of the DataType, such as ITEMHEADER and ITEMVALUE, create an object of the class.
  - Find the DDA part (from the “MBDDA” to “MEDDA”);
  - Find the occurrence of values (from end of “MEDDA” to the next beginning of “MBDDA”), the values (maybe multiple occurrences) are put into a vector.
  - Call the corresponding functions of bodReader class for each segment of the attribute in a data segment.
  - Processing the definition and the corresponding values.
  - Attach it to the parent DDA according to the specific information about each BOD.
  - If there is any DDA left, go to loop.
- End;

SECTION 4
BOD VALIDATION AND CONVERSION

The run-time facilities of the BOD modeling and validation tool suite are described in this section. The sections are arranged to follow the process order of transmitting a BOD. It starts from the BOD creation in a source application and ends with the BOD consumption by a target application. We use SYNC_ITEM as the example BOD to show the process. The valueForLotsnflag rule of ITEMHEADER is used to illustrate the activation of ECAA rules for BOD validations.

4.1 Population of BODs Using the Modified Source Skeletal Program

The generated program will have a main function inside the class. Actually, it is the only public function of the class. The function creates an instance of SYNC_ITEM which is the top-level class. CNTROLAREA and SENDER instances are created later on. Several "set" method calls are made on the SENDER object. The last portion of the program is to set attributes for ITEMHEADER. Since SYNC_ITEM may contain many instances of ITEMHEADER, an array of ITEMHEADER is created.

As we mentioned in Section 3, skeletal program must be modified by the application vendors to fit their needs. For example, the constants "component" for component fields, "task" for task fields, etc should be replaced by program variables. The value for "LOTSNFLAG" must be changed in order to meet its data constraint.

4.2 Activation of ECAA Rules to Enforce the Source Constraints

The enforcement of source constraint is automatic in the sense that a programmer does not have to explicitly call the method to enforce it. At build-time, a call to the fireRule function of a rule class is automatically inserted into the implementation of a set function. The call to the set function for LOTSNFLAG in the skeletal program is made using itemheader2[itemheader2_L].setLOTSNFLAG("lotsnflag").

If the application vendor forgets to replace "lotsnflag" with one of the legitimate values ("L", "S", "N" and "B" in this case), the following error message will appear:

```
Within rule SYNC_ITEM_SCHEMA::RCvalueForLotsnflag
Value for LOTSNFLAG must be S, L, N or B!
```

Figure 4.1 Error Message when a Rule Is Violated

The implementation of setLOTSNFLAG includes three steps. First, we create an instance of RcvalueForLotsnflag. Second, a call to fireRule method of the object is made. Third, obj is assigned to LOTSNFLAG using the assignment statement of the Java language.

4.3 Conversion to Physical BOD

Conversion to the physical BOD is done by the source converter, which transforms the populated Java class instances into a physical BOD. For an example of a physical BOD, see Figure 3.5 of Section 3.

4.4 Transfer of Physical BODs between Source and Target Applications

The communication infrastructure should provide an API interface for our facilities to transmit BODs at run-time. On the source side, when the physical BOD is generated by the source application system, a function call to an API is made to delivery BOD to the infrastructure. On the target side, the infrastructure makes a call to the adapter and passes the BOD to the adapter. Once the BOD is available, the target converter is used to convert the physical BOD into Java class instances. The communication infrastructure can be CORBA, DCOM, RMI or others.

4.5 Conversion of Physical BOD into Java Class Instances

Conversion of a physical BOD into Java class instances is done by the target converter. Within the generated code, the array fldnames is used to hold all the attribute names defined in a DDA (Data Definition Area). Each name is compared with all the possible names such as QUANTITY, ITEM, and ITEMTYPE. If there is a match, a substring from position dp to position dp + len[i] of str is used to "set" the value for the attribute. If the type for the attribute is not string, a type conversion is carried out.

4.6 Activation of ECAA Rules to Enforce the Target Constraints

The target constraints are enforced by the activation of ECAA rules. The process is almost the same as the enforcement of source constraints on the source side. Therefore, we shall not repeat the procedure here.

4.7 Accessing Data Using the Modified Target Skeletal Program

In the generated code, a variable is used to hold the result of a "get" method. For example, we use the statement of the form "String verb10 = cntrolarea2.getVERB()" to get the verb of the BOD. In this example, the suffix 10 of verb is automatically generated, and cntrolarea2 is the object for CNTROLAREA. The number 2 after cntrolarea is also automatically generated. We generate a statement to print out a string or an integer value if it is not null.

4.8 Moving Data into the Target Application System

Now that data have been retrieved from the Java classes using "get" methods, it can be moved into the target system by calling the APIs provided by an application

system. Once the data have been moved into the application, the application can do any further processing on them.

SECTION 5
SUMMARY, DISCUSSION AND FUTURE WORK

5.1. Summary

In this report, we have developed program facilities for modeling and validating BODs being transmitted among application systems. XGTOOLS, which is a graphical user interface tool running on AIX machines, is used to model BODs as entity classes having keyword constraints and ECAA rules. The entity classes are translated into EXPRESS+ scripts and dictionary interchange files. The former is generated for users' reading whereas the latter is generated for populating the metadata repository running on NT machines. Keyword constraints are translated into ECAA rules so that the enforcement of constraints and rules can be handled in a uniform way. Based on the metadata stored in the metadata repository, JCLIs, converters and skeletal programs are automatically generated.

The task of validating BODs is done by the enforcement of ECAA rules at run-time. Since the generated JCLIs contain Java class implementations of ECAA rules, these rules are enforced when an application system generates a new BOD to be sent to other application systems or when an application system receives a BOD from another application system. Thus, BODs are automatically validated on both the source side and the target side based on the conceptual models of the same BOD defined for both sides. The validation is done outside of the application systems, therefore requires no changes to them.

5.2. Discussion

5.2.1 Features

The approach taken in this work allows BODs to be validated to ensure that they satisfy not only the constraints as specified by OAG, but also the constraints as viewed by the source and target applications. It will avoid errors that can occur in BOD transfers among application systems if no such validations are carried out. This approach also allows the application systems to move data in their native data types without first converting all the data items into strings and read data into them using their native data types. The creation and transfer of physical BODs are transparent to programmers.

In this system, ECAA rules are used to capture constraints of BODs at build-time and validate these constraints at run-time. In our approach, each ECAA rule is translated into a corresponding Java class. In addition to Java Call Level Interfaces, skeletal programs are also generated to ease the use of JCLIs by the programmers of application systems. Since the object-oriented approach is taken to model BODs and to do code generation, we believe that the generated skeletal programs are easy to understand, modify and use.

5.2.2 Limitations

The set of tools for BOD modeling and validation is divided into two parts. The modeling tools take advantage of some existing graphical tools (i.e., XGTOOLS). They are written in C and run on AIX machines and the rest of the tool set are written in Java and run on NT machines. Metadata of BODs captured in XGTOOLS are translated into dictionary interchange files and are transferred to NT machines via ftp. It will be better if they are all written in Java so that they can all run on all kinds of computing platforms. However, the redevelopment of XGTOOLS in Java is a non-trivial task. We feel that it is important to support all sorts of object modeling tools that can potentially be used to model BODs. The metadata captured by these tools can be converted into the dictionary interchange form, which can be imported into the metadata repository (see further discussion in Section 5.3.2).

Some readers may feel that our translation of ECAA rules into rule code at build-time is a limitation because the approach does not support "dynamic rules". However, it is our belief that constraints associated with BODs are rather static. Constraints of BODs specified by OAG are fixed since they are presumably standard specifications. Additional constraints associated with BODs as specified by application systems are low-level data constraints. They are not like business rules and policies, which are more likely to change to suit the dynamic nature of business. If BOD constraints are subject to frequent changes, then a dynamic way of implementing ECAA rules will be needed. The technique for implementing dynamic rules is available. For example, when a rule is modified at run-time, the generated Java class for the rule can be used to replace the class for the old rule. Thus, the new rule will be used for BOD validation the next time the rule is triggered. The detailed description of the dynamic rule management is out of the scope of this report.

5.3 Future Work

5.3.1 Enhancement to XGTOOLS

One possible task is to enhance the XGTOOLS to make them general-purpose graphical tools. They can then be used to model not only BODs but also other types of objects, which represent data entities used by the components of an integrated systems (e.g., agents, legacy application systems, etc.) as well as the component systems themselves. Additional constraint types need to be supported by the modeling tools.

5.3.2 Commercial Object-oriented Modeling Tool

It will be ideal if the developed BOD modeling and validation system can accommodate some commercial object-oriented modeling tools such as Rational Rose. They can be used for modeling the structures and behaviors of BODs and other types of objects as well as some constraints associated with them. Constraints not captured by commercial tools can be captured by ECAA rules. Constraints captured by these tools can be translated into ECAA rules as we have demonstrated in our work. They can then be translated into Java classes for run-time enforcement of constraints. The use of commercial modeling tools will be better than XGTOOLS since they are supported by

the vendors of these tools. Metadata captured by these tools can be translated into our dictionary interchange form and be imported into the metadata repository.

5.3.3 Information Mediation

In an integrated system consisting of a number of heterogeneous application systems, the data representations in these applications are bound to have some naming, syntactic and semantic differences. For example, synonyms and homonyms can exist in naming attributes and data entities; an attribute of one system can be a data entity of the other system; the values X, Y and Z of an attribute in one system can be the same as A, B and C of another system. Data with different representations need to be "mediated" by converting them from one data representation to another.

In case of BOD transfer, a BOD can be created by one application system and be sent to multiple application systems. The different data representations between each pair of the source application and the target application need to be explicitly specified. It will be more efficient if the mediation tasks can be carried out in parallel in the target systems that receive a BOD based on their specified differences. In our architecture for BOD validation, the target converter, which convert a physical BOD into Java class instances can be extended to do the mediation task. Mediation specifications stored in the form of, say, a data mapping table can be provided to each converter. When the converter picks up a data value in string format from a physical BOD, it can look up the mapping table to see if the value needs to be converted to meet the target system's specification. If it needs to be, the value will be converted and used as the value of the corresponding attribute of a Java class. Thus, when the target application accesses the attribute value from the Java class, the value is in the representation needed by the target system. The above example is for resolving the semantic difference of an attribute value. A similar approach can be taken to deal with naming and syntactic differences. Since multiple target converters are automatically generated for different application systems to receive BOD data, the approach described above is a distributed approach to handle the mediation problem. It can be much more efficient than a centralized approach in which all mediation tasks are carried out by a single information mediator.

APPENDIX

BNF FOR EXPRESS+

The method and knowledge specifications are defined. The knowledge specifications include the definition of ECAA rules.

```
EXPRESS+ := SCHEMA_PART METHOD_PART KNOWLEDGESPEC_PART

SCHEMA_ART := “normal EXPRESS definition part”

METHOD_PART := “METHODSPEC” SPEC_NAME “in” SCHEMA_NAME
RETURN_TYPE METHOD_NAME “(“ PARAMERT_LIST “)”
“END_METHODSPEC”

KNOWLEDGESPEC_PART := “KNOWLEDGESPEC” SPEC_NAME “in” SCHEMA_NAME
ECAA_blocks
“END_KNOWLEDGESPEC”

SPEC_NAME := Valid_Identifier
SCHEMA := Valid_Identifier

ECAA_blocks := (EVENT_DEFINTION RULE_DEFINITION TRIGGER_DEFINITION
RULE_GROUP_DEFINTION)*

EVENT_DEFINITION := “EVENT” EVENT_NAME “(“ PARAMETER_LIST “)”
“CouplingMode = ” MODE
“TriggerMethod = “ CLASS_METHOD
“END_EVENT”

MODE := “AFTER” | “BEFORE”

CLASS_METHOD := CLASS_NAME “::” METHOD_NAME

CLASS_NAME := Valid_Identifier
METHOD_NAME := Valid_Identifier


RULE_DEFINITION := “RULE” RULE_NAME
“TYPE =” TYPE
“STATE =” STATE
“CONDITION =” CONDITION
“ACTION = “ ACTION
(“ALTACTION = “ ALTACTION)
“END_RULE”


CONDITION := EXPR
```

```
EXPR := SIMPLE_BOOLEAN_EXPR | SIMPLE_BOOLEAN_EXPR OPR EXPR

OPR := "AND" | "OR" | "NOT"

SIMPLE_BOOLEAN_EXPR := Valid_Identifier REL_OPR Valid_Identifier

REL_OPR := "<" | ">" | ">=" | "<=" | "=" | "!="

ACTION := ASSIGNMENT | METHOD_CALL
ALTACTION := ASSIGNMENT | METHOD_CALL

ASSIGNMENT := VARIABLE "=" VALUE

VARIABLE := Valid_Identifier

VALUE := Literal | VARIABLE

Valid_Indentifier := EXPRESS_Identifier

Literal := EXPRESS_Literal

METHOD_CALL := OBJ_NAME "." METHOD_NAME "(" VALUE_LIST ")"

RULE_NAME := Valid_Identifier
TYPE := "STATIC" | "DYNAMIC"
STATE := "ACTIVE" | "INACTIVE"


TRIGGER_DEFINITION := "TRIGGER" TRIGGER_NAME
"EVENT = " EVENT_NAME
"RULE_SET = " "(" RULE_LIST ")"
"END_TRIGGER"

RULE_LIST := RULE_NAME ("," RULE_NAME)*


RULE_GROUP_DEFINITION := "RULE_GROUP"
"RULE_SET = (" RULE_LIST ")"
"END_RULE_GROUP"
```

LIST OF REFERENCES

[ACT96] ACT-NET Consortium, “The Active Database Management System Manifesto: A Rulebase of ADBMS Features,” ACM SIGMOD Record Vol. 25, No. 3, September 1996, pp. 40-49.

[BUC95] Buchmann, A.P., Zimmermann, J., Blakeley, J.A., and Wells, D.L., “Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions”, Proceedings of IEEE International Conference on Data Engineering, Taipei, Taiwan, 1995, pp. 117-128.

[CHA94a] Chakravarthy, S., Anwar, E., Maugis, L., and Mishra, D., “Design of Sentinel: an Object-Oriented DBMS with Event-Based Rules”, Information and Software Technology, Vol. 39, No. 9, London, September 1994, pp. 555-568.

[CHA94b] Chakravarthy, S., Mishra, D., “Snoop: An Expressive Event Specification Language for Active Databases”, IEEE Data and Knowledge Engineering Vol. 14, No.1, 1994, pp. 1-26.

[CHU96] Chu, B., Tolone, W. J., Wilhelm, R., Hegedus, M., Fesko, J., Finin, T., Peng, Y., Jones, C., Long, J., Matthews, M., Mayfied, J., Shimp, J., and Su, S., “Integrating Manufacturing Softwares for Intelligent Planning-Execution: A CIIMPLEX Perspective”, Plug and Play Software for Agile Manufacturing, Boston, MA. Proceedings of SPIE Vol. 2913. 1996, pp. 96-108.

[DAY88] Dayal, U., Blaustein, B., Buchmann, A.P., Chakravarthy, U., Hsu, M., Ledin, R., McCarthy, D., Rosenthal, A., Sarin, S., Carey, M., Livny, M., and Jauhari, R. “The HiPAC Project: Combining Active Databases and Timing Constraints”, ACM SIGMOD Record, Vol. 17, No. 1, 1988, pp.51-70.

[DOW98] Downing, T., Java RMI: Remote Method Invocation, IDG Book Worldwide, Foster City, CA, 1998.

[EDD98] Eddon, G., and Eddon, H., Inside Distributed COM. Microsoft Press, Redmond, WA, 1998.

[GAR95] Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J., "Integrating and Accessing Heterogeneous Information Sources in TSIMMIS". Proceedings of the AAAI Symposium on Information Gathering, Stanford, CA, March 1995, pp. 61-64.

[GEH91] Gehani, N.H. and Jagadish, H.V., "ODE as an Active Database: Constraints and Triggers", Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona (Catalonia, Spain), September 1991, pp. 327-336.

[GEH96] Gehani, N.H., Lieuwen, D.F., and Arlein, R., "ODE Active Database: Trigger Semantics and Implementation", Proceedings of International Conference on Data Engineering, 1996, pp. 412-420.

[HAN92] Hanson, E.N., "Rule Condition Testing and Action Execution in Ariel", Proceedings of 1992 ACM SIGMOD Conference on Management of Data, San Diego, CA, June 1992, pp. 49-58.

[HAN93] Hanson, E.N., and Widom, J., "An Overview of Production Rules in Database Systems", The Knowledge Engineering Review, Vol. 8, No. 2, 1993, pp.121-143.

[HAR97] Hardwick, M., and Bolton, R., "The Industrial Virtual Enterprise" Communications of the ACM, Vol. 40, No. 9, 1997, pp. 59-60.

[INT94] International Organization for Standardization, ISO 10303-11 Industrial Automation Systems and Integration – Product Data Representation and Exchange – Description Methods: The EXPRESS Language Reference Manual, 1994.

[LAM92] Lam, H., and Su, S.Y.W., "GTOOLS: An Active Graphical User Interface Toolset for an Object-oriented KBMS", International Journal of Computer Science and Engineering, Vol.7, No. 2, April 1992, pp. 69-85.

[LOH91] Lohman, G.M., Lindsay, B., Pirahesh, H., and Schiefer, K.B., "Extensions to Starburst: Objects, Types, Functions, and Rules," Communications of the ACM, Vol. 34, No. 10, October 1991, pp. 94-109.

[OBJ92a] Object Management Group, Object Management Architecture Guide, John Wiley & Sons, Inc., New York, September 1992.

[OBJ92b] Object Management Group, The Common Object Request Broker: Architecture and Specification, John Wiley & Sons, Inc., New York, 1992.

[OPE95] Open Applications Group "Business Object Documents", available at http://www.openapplications.org, 1995

[PAP95] Papakonstantinou, Y., Garcia-Molina, H., and Widom, J. "Object Exchange Across Heterogeneous Information Sources", IEEE International Conference on Data Engineering, Taipei, Taiwan, March 1995, pp. 251-260.

[PAT93] Paton, N.W., Diaz, O., and Barja, M.L., "Combining active rules and metaclasses for enhanced extensibility in object-oriented systems", Data and Knowledge Engineering, Vol. 10, 1993, pp. 45-63.

[SCH91] Schreier, U., Pirahesh, H., Agrawal, R., and Mohan, C., "Alert: An architecture for transforming a passive DBMS into an active DBMS," Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona (Catalonia, Spain), September 1991, pp. 469-478.

[SHY91] Shyy, Y.M., and Su, S.Y.W., "K: A High level Knowledge Base Programming Language for Advanced Database Applications," Proceedings of 1991 ACM SIGMOD Conference on Management of Data, Denver, CO, May 1991, pp. 338-347.

[SHY96] Shyy, Y.M., Arroyo, J., Su, S.Y.W., and Lam, H., "The Design and Implementation of K: A High-Level Knowledge-Base Programming Language of OSAM*.KBMS", Very Large Data Base (VLDB) Journal, Vol. 5, No. 3, 1996, pp. 181-195.

[STO91] Stonebraker, M., and Kemnitz, G., "The Postgres Next-Generation Database Management System", Communications of the ACM, Vol. 34, No. 10, October 1991, pp.78-92.

[SU89] Su, S.Y.W., Krishnamurthy, V. and Lam, H., "An Object-oriented Semantic Association Model (OSAM*) for Modeling CAD/CAM Databases", AI in Industrial Engineering and Manufacturing: Theoretical Issues and Applications, Kumara, S. and Kashyap, R.L. (eds.), American Institute of Industrial Engineers, Industrial Engineering and Management Press Norcross, GA, 1989, pp. 463-494.

[SU90] Su, S.Y.W., and Lam, H., "Object-oriented Knowledge Base Management Technology for Improving Productivity and Competitiveness in Manufacturing", Proceedings of the 4th Conference on Design and Manufacturing Systems Research, Arizona State University, Tempe, AZ, January 8-12, 1990, pp. 161-167.

[SU91] Su, S.Y.W., and Alashqur, A., "A Pattern based Constraint Specification for Object-oriented Databases", Proceedings of COMPCON, Spring 1991.

[SU92] Su, S.Y.W., and Lam, H., "An Object-Oriented Knowledge Base Management System for Supporting Advanced Applications", Proceedings of the 4th International Hong Kong Computer Society Database Workshop, December 12-13, 1992, pp. 3-21.

[SU93] Su, S.Y.W., Guo, M., and Lam, H., “Association Algebra: A Mathematical Foundation for Object-oriented Databases”, IEEE Transactions on Knowledge and Data Engineering, Vol. 5. No. 5, October 1993, pp. 775-798.

[SU95] Su, S.Y.W., Lam, H., Arroyo-Figueroa, J.A., Yu, T.F., and Yang, Z., “An Extensible Knowledge Base Management System for Supporting Rule-based Interoperability among Heterogeneous Systems”, Proceedings of the Conference on Information and Knowledge Management (CIKM '95), Baltimore, MD, November 28 - December 2, 1995, pp. 1-10.

[SU96a] Su, S.Y.W., Lam, H., Yu, T.F., Arroyo-Figueroa, J.A., Yang, Z., and Lee S., “NCL: A Common Language for Achieving Rule-Based Interoperability among Heterogeneous Systems”, Journal of Intelligent Information Systems, Special Issue on Intelligent Integration of Information, Vol. 6, 1996, pp. 171-198.

[SU96b] Su, S.Y.W., and Lam, H., “Enterprise Rule Management and Services”, internal project design document, Database Systems Research and Development Center, University of Florida, Gainesville, FL, 1996.

[SUN] Sun Microsystems, Java Compiler Compiler, available at http://suntest.sun.com/JavaCC, 1997.

[WID90] Widom, J., and Finkelstein, S. A., “Set-Oriented Production Rules in Relational Database Systems”, Proceedings of 1990 ACM SIGMOD Conference on Management of Data, Atlantic City, NJ, May 1990, pp 259-270.