An Extensible, Ontology-based, Distributed Information System Architecture

Alan I. Chao
Basil C. Krikeles
Angela E. Lusignan
ALPHATECH Inc.
6 New England Executive Park
Burlington, MA 01803 USA.
Alan.Chao@alphatech.com Basil.Krikeles@alphatech.com
Angela.Lusignan@alphatech.com

Edward Starczewski
AFRL/IFEA
32 Brooks Road
Rome, NY 13441, U.S.A.
Edward.Starczewski@rl.af.mil

Abstract - *The AFRL-sponsored Adaptive Sensor Fusion (ASF) program produced a software framework, the eXtensible Distributed Architecture (XDA), which facilitates the construction of scalable, flexible distributed systems. XDA is based on a simple ontology mechanism that enables the definition and maintenance of high-level object models to capture the shared semantics necessary for interoperability. This ontology-based approach strikes a balance between having enough expressiveness to capture complex semantic interactions between components and being succinct enough for efficient, cross-platform implementation. Shared object models can evolve and can be refined incrementally through standard, object-oriented reuse mechanisms of inheritance and containment. XDA has been applied to sensor fusion and visualization. Additionally, XDA is being used to construct a database abstraction layer to access multiple, heterogeneous databases. XDA ontology objects can be persisted and can be retrieved with location transparency using an object-based query language. This capability allows objects residing in shared resources to be accessed uniformly by collaborating applications in a distributed environment.*

Keywords: Ontology, domain model, system evolution, distributed system.

1 Introduction

Achieving interoperability in a distributed environment is notoriously difficult because most systems are designed to operate in isolation using their own specialized views of a problem domain. Furthermore, once such systems are built, they are typically difficult to maintain and to extend due to the complexity caused by semantic dependencies between components as well as *ad hoc* semantic conventions. In this paper we present an ontology-based framework (eXtensible Distributed Architecture or XDA) that addresses many of the problems associated with the construction of complex distributed information systems.

We will provide a high-level overview of the philosophy and design of XDA with emphasis on ontology support, followed by a description of some concrete problem domains with associated ontology-based approaches to handling them. A detailed description of the Adaptive Sensor Fusion program and XDA has appeared in [1].

2 XDA design overview

We use the term "ontology" in a very restricted sense to refer to XDA's approach for data definition and structuring with foundational support for single inheritance and aggregation. XDA has a lightweight mechanism for domain engineering. There is no requirement to define a universal ontology. XDA encourages local solutions to the problem of domain engineering in terms of sets of related structured data types. We call these sets of data types *ontologies*. Each data type within an ontology is called an *entity* and is defined in terms of a set of typed attributes. Partial semantic overlap as well as semantic inconsistencies between these domains may exist and when they exist they are resolved programmatically. There is explicit support for reuse of existing entity definition across different ontologies using both inheritance and aggregation. Our use of the term *ontology* is therefore highly restricted and should not be confused with more complex constructs such as knowledge bases.

XDA addresses the difficult problem of distributed application development, yet it is lightweight, simple, robust, easily extensible and capable of high performance. By factoring out the domain semantics from the application layer it enables disciplined system evolution. Furthermore, XDA automatically generates an implementation of the domain model both in terms of C++ and Java, making it especially easy to propagate object model changes to the applications. Thus, XDA ontologies are automatically translated to object models shared across applications and defining the semantics of inter process communication. Objects can be transported across applications with a variety of inter process communication technologies (such as CORBA or Web Services). Finally, the object model supports persistence

with location transparency through a vendor neutral, object oriented database abstraction layer. For a detailed description of XDA's persistence capability see [2].

2.1 Object model design and deployment

In broad terms, information processing for a specific problem domain consists of a set of applications (implementing what is sometimes referred to as business logic) that operate on a set of structured data types (domain model). Processing may involve instantiating, mutating, persisting, retrieving or transporting instances of these data types. When processes collaborate they need to share at least part of the domain model. When this sharing of semantic information is not carefully managed it can result in tight coupling between components and in a system that is difficult to extend and maintain. The amount of coupling can increase proportionally to the square of the number of applications involved. XDA mitigates this problem with a methodology for defining a shared object model (ontology) that is well-behaved with respect to inheritance and aggregation thus allowing the designers to better control coupling between components.

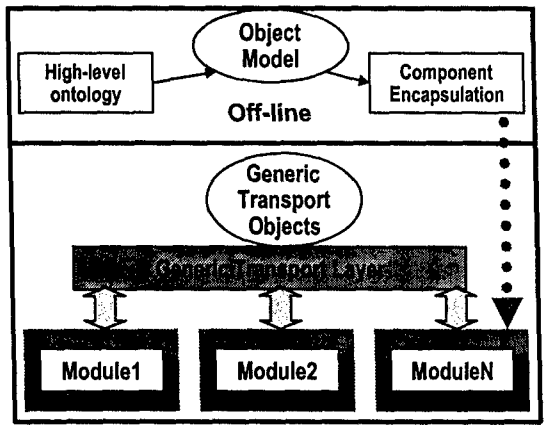

Figure 1. Object model definition and deployment

As shown in Figure 1, building an XDA-based information system involves two broad activities: 1) designing an object model which is appropriate for the problem domain and rich enough to capture the semantics required for information processing and 2) building applications that can interoperate in terms of the shared domain model and that supply the needed processing logic. In order to maximize interoperability, XDA defines a common set of public interfaces that are implemented by all participating applications. An application that implements these interfaces and is capable of utilizing the shared object model is called *encapsulated.* Such applications can be controlled at runtime in a uniform manner, and can be used as building blocks for constructing distributed applications. XDA includes a software developer's toolkit for building encapsulated applications.

2.2 Encapsulation Methodology

As noted in Section 2.1, XDA consists of an encapsulation library with interfaces that are designed to facilitate both building new components and encapsulating existing components. The latter is extremely important since it provides a way to utilize existing components. Any component following this standardized approach can then interoperate with other XDA components seamlessly. The figure below illustrates the architecture of an encapsulated legacy component:

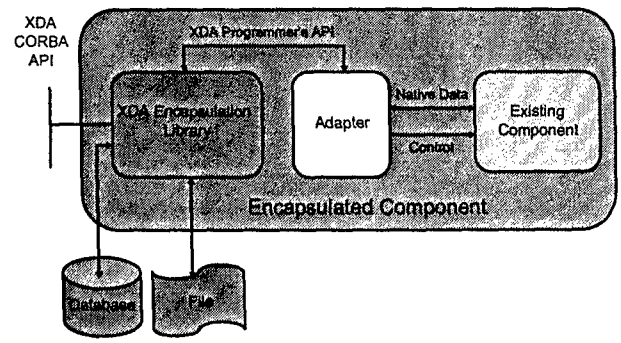

Figure 2. Integration of Legacy Components

Wrapping an existing application within XDA involves building a module that mediates between the existing application and XDA. This is illustrated as the *Adapter* in the figure above. In addition to configuring and controlling the existing component, the adapter also translates between ontology-based data and native data structures that are required by the legacy component. The adapter uses the *XDA Programmer's API* to communicate with the *XDA Encapsulation Library* that supports inter process communication using publish/subscribe, file or database. This design insulates the adapter implementation from all vender dependencies, including middleware and database software.

2.3 Model-on-a-model architecture

A major design goal of XDA is the ability to modify the shared object model without having to update the low-level transport infrastructure. XDA supports domain model evolution without changes to the transport mechanism, by enforcing a strict separation between high-level domain definition and low-level concerns such as persistence and inter process communication. Under XDA it is possible to add new entities to an ontology without modifying the physical database schemas. This is accomplished through a "model-on-a-model" architecture-flexible, high-level, semantically rich domain models are implemented in terms of a fixed, generic object model for transport and persistence.

The figure below illustrates the "model-on-a-model" architecture of XDA.

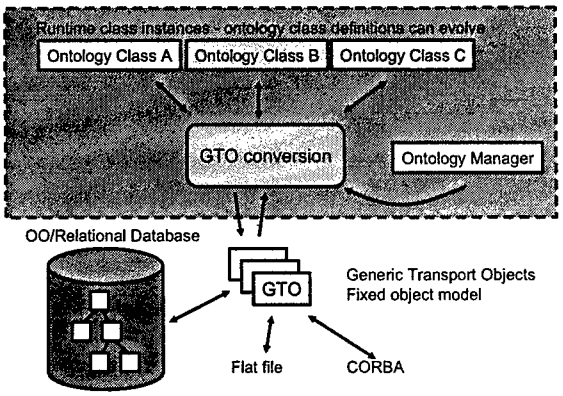

Figure 3. XDA has a *model-on-a-model* architecture

There is a one-to-one correspondence between Ontology Objects and Generic Transport Objects (GTOs) in the fixed low-level object model. The difference is that the Ontology Objects carry the semantics associated with their domain model. GTOs contain no domain specific information, and capture only the raw data and structure information from the original object. Each GTO contains a reference to its ontology, and combined with that ontology it can reproduce an instance of the original object. When ontology class instances need to be transported, they are converted to GTOs. These GTO instances can be persisted to a database, saved to a file, or transported across platforms as byte-streams. Their object structure information combined with the ontology-based semantics can be used to implement efficient ontology-driven queries that can be used for data filtering.

2.4 Ontology definition

An XDA ontology is a means of describing structured (hierarchical) data; there are no associated methods or predicates within the ontology definition itself. An *ontology* consists of a set of *entities* and a set of *enumerated* types. Each entity contains a set of *attributes* that have a *name* and *data type*. An entity can be derived from a *base entity* (single inheritance), and inherit the base entity's attribute set. The base entity is specified by its name and by the name of its ontology. Supported attribute data types include the basic types (integer, double, and string) and sequences of those basic types. Additionally, the data type of an attribute can be *user defined* or *user defined sequence*. When an attribute is *user defined* its value is an instance of another entity that has been defined either within the same ontology or within a different ontology. Similarly, an attribute with type *user defined sequence* has as a value a sequence of another entity. Finally, the data type of an attribute could also be a *user defined enumerated type*.

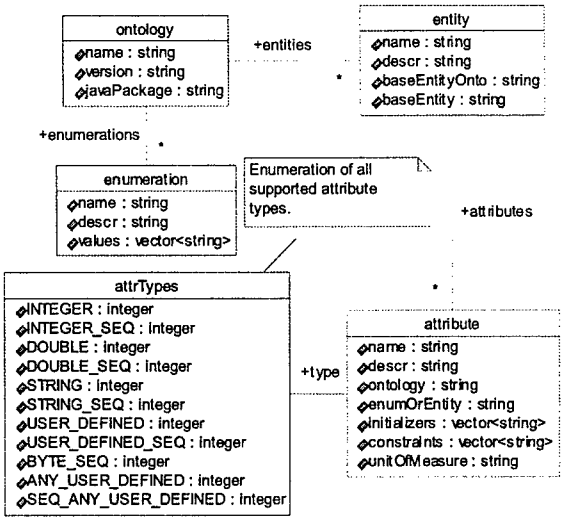

Figure 4. UML Diagram for Ontology

The containment hierarchy of an ontology object is a tree structure and can be directly compared to the structure of XML text. Given this similarity, XML can be used as an alternate transport format for ontology objects. However, GTO streams are binary and do not involve ASCII conversion overhead, allowing for better optimization strategies.

2.5 Automatic code generation and GTOs

XDA's code generator converts ontology definitions to fully implemented application classes in C++ and Java. For each entity defined within an ontology, a C++ class with corresponding header and implementation files or a Java class file is automatically generated. This generated code can be incorporated into the application component that uses the specified ontology.

Included in the class interface are methods for mediating conversion to and from the GTO format, as well as accessor and mutator methods for the attributes of the class. In addition, the class supports virtual construction of class instances from GTO objects. XDA has the capability to invoke the correct constructor for the object given its GTO. This capability is used recursively when objects within the containment hierarchy are constructed. It should be noted that substitutability applies to sequence valued attributes and sequence valued attributes are in general heterogeneous. If the attribute type is *sequence of A*, and if B derives from A, then the sequence attribute could contain instances of both B and A. XDA guarantees that all objects within a root object will be constructed correctly according to their exact type.

3 Current XDA Ontologies

3.1 XDA system ontologies

XDA uses the ontology mechanism in implementing its own capabilities such as dynamic distributed tasking, data querying, messaging services, and resource management. Provided in the following sections are

illustrations for some of these system level ontologies and descriptions of their use within a distributed system.

3.1.1 Meta Ontology

XDA ontologies are self-describing in the following sense: there is an ontology for ontologies providing the definition of what an ontology is. (Shown in Figure 4) This *meta* ontology results in significant advantages including uniform treatment of instance data (ontology data are treated identically to other data for transport and persistence), and economy of implementation for maintainability. The *meta* ontology; however, is special because it serves as a bootstrap ontology - an ontology that is needed in order to define what ontology means. Nevertheless, it is still possible to enhance the *meta* ontology without breaking a deployed system and it is possible to automatically generate code for refined versions of the meta ontology.

3.1.2 System Message Ontology

Encapsulated applications can generate simple feedback messages to XDA system monitoring tools or other components using the System Message Ontology. When information products match a subscribing component's criteria and are successfully published, the XDA encapsulation layer generates a *sys pub notify* message. These publish notify messages provide high-level information about the transmitted data. These messages can be used to track component subscriptions and provide useful information regarding when the data was published and to whom the data was published.

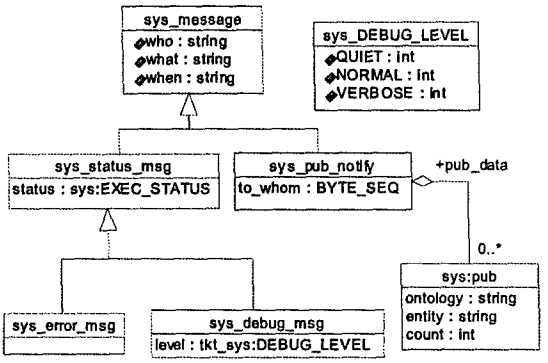

Figure 5. System Message Ontology

Each process within a distributed application emits messages at runtime. Certain sets of these messages can be aggregated to form patterns that correspond to *complex* events having special meaning for the distributed system as a whole. These patterns of messages are useful both in order to detect *exceptional* conditions for the system as a whole, as well as to detect *normal aggregate events* within the system. An example of such an aggregate event is task startup, described in more detail in Section 3.1.3.

Exceptional conditions are particularly difficult to handle in a distributed system. For example, an exceptional condition in a distributed system may be an aggregation of conditions in a number of applications where each individual condition is not necessarily an error.

Detection of these conditions requires a status monitoring application that is capable of aggregating multiple messages into complex messages that give views of the system at a higher level of abstraction.

3.1.3 Task Ontology

XDA supports the execution of complex distributed tasks through the definition of the Task Ontology (Shown in Figure 6). This ontology can be used to define hierarchical tasks since the *task* entity is self-referential. The leaf nodes in this tree structure are *jobs* representing processes that run on a specific host processor. In a CORBA environment the process *name* is the *Naming Context* of the process and it can be used to connect to or activate the process using a CORBA Naming Service. Each task also contains additional attributes that are used for scheduling task execution, setting the task priority, and storing the tasks status.

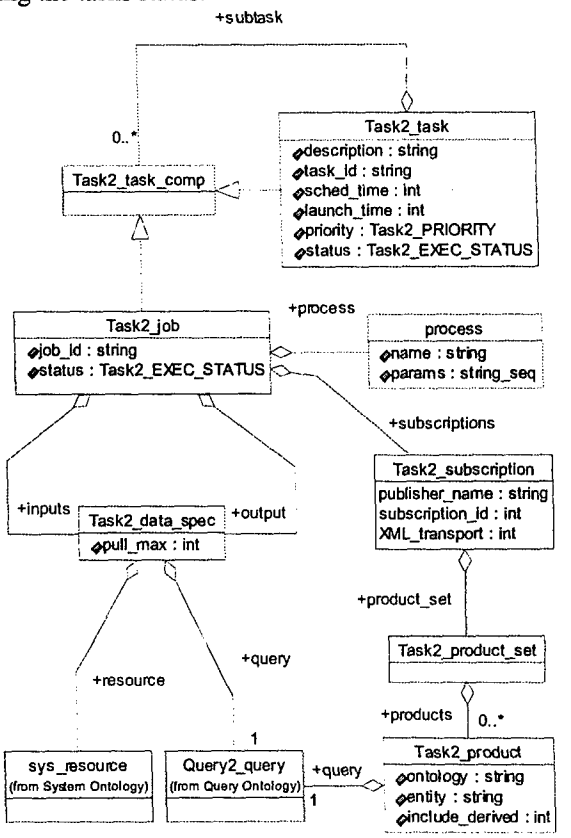

Figure 6. Task Ontology

Each job contains a set of input and a set of output data specifications. Each data specification is associated with a distributed (shared) system resource, and a query that is used for filtering the data both on the input and output specifications. Shared resources can be files or databases.

Finally, each job contains a set of subscriptions for input data. The subscription set is part of XDA's peer-to-peer publish/subscribe mechanism. The *publisher name* of a subscription is again the *Naming Context* of the publisher, allowing the job to connect to the publisher and invoke an appropriate message that sets up a subscription. Each subscription is for a specific product set, containing a product type expressed as an ontology-entity pair with an associated object-based query to further refine the products relevant to this subscription. The ability to precisely define the product set of a subscription allows the task developers to finely control the bandwidth requirements of the subscriptions. Another interesting feature of subscriptions is that they can be sensitive to inheritance and substitutability - a product set can be configured to include derived entities.

In order to control distributed task execution, XDA includes a Task Manager application whose purpose is to execute, monitor, and shut down tasks. As shown in the figure below, end-users can modify instances of the task ontology though XDA graphical tools. Users can add new jobs to the task, change the inputs and outputs of data specifications, and modify subscriptions to existing components without rebuilding their systems. These task objects are then sent to the task manager whose main responsibility is to execute and monitor the processes specified within the task.

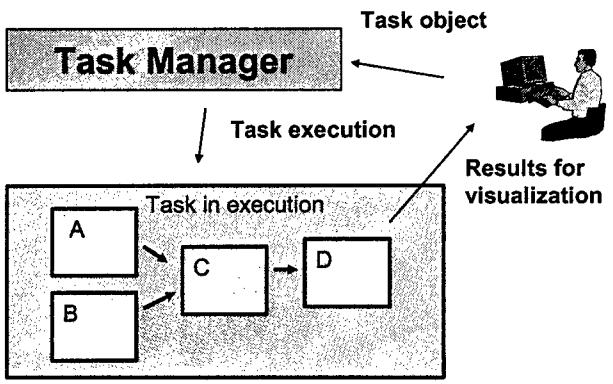

Figure 7. The Task Manager orchestrates task execution

The Task Manager provides interfaces that permit users to dynamically replace system components at runtime. For example, a user is able to replace component C by another component E without shutting down components A, B and D.

The Task Manager is also capable of message aggregation to detect task-wide conditions of interest. An example of such an aggregate event is task startup. When a user submits a task for execution, the Task Manager activates all jobs that are in the task. It starts monitoring these jobs for status messages. Before declaring that the task has successfully started, the Task Manager must receive messages from each participating job that it has successfully started.

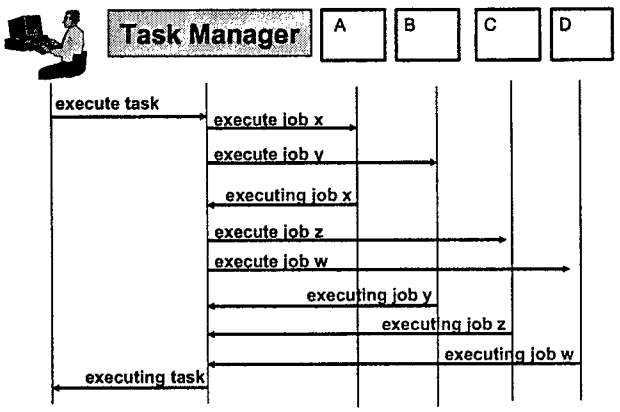

Figure 8. Task startup is a complex event

3.1.4 Query Ontology

XDA provides an object-based database abstraction layer that decouples applications from the details of how objects are physically laid out, and eliminates vendor-specific dependencies. This design combines with the "model on a model" architecture and substitutability resulting in the following benefits: (1) object models can evolve with changes easily propagated to applications, (2) applications can be updated incrementally to the latest object model changes, (3) object models can evolve without affecting the database code, (4) object based queries can remain valid if object models evolve based on inheritance and aggregation, (5) database applications can be replaced without making changes to XDA applications, (6) multiple database applications can be used simultaneously (for example MySQL and Oracle), and (7) multiple database technologies can be used simultaneously (for example relational and object oriented). The XDA database abstraction layer has been implemented in terms of JDBC, ODBC, direct SQL and the object-oriented database ObjectStore.

XDA queries are expressed in terms of the high-level ontology definition of the problem domain rather than the low-level database schema. This is more meaningful to the end-user since ontologies are used to define the domain of interest. Under this design, end-user or application queries can remain valid through database implementation or vendor changes.

An XDA query can be applied to any sequence of Generic Transport Objects. Since GTOs are objects, XDA queries are based on traversal paths down the containment hierarchy of the Generic Transport Object. Consistent with the XDA strategy, the query entity is defined in terms of a Query Ontology. A UML diagram of the Query Ontology is shown below:

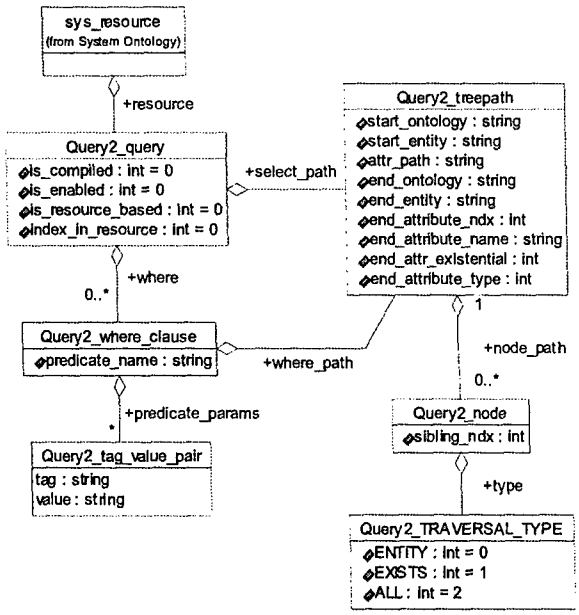

Figure 9. Query Ontology

Each query contains a *select path* that permits the selection of a contained entity or entities within the containment hierarchy, and zero or more *where clauses*. Each where clause consists of a path continuing from the select path that leads to an entity farther down the containment hierarchy, and a *predicate* that can be applied to an entity or attribute to further serve as a selection criterion.

Figure 10 shows three sample queries based on the simple target hypothesis ontology (Shown in Figure 11). The first query uses the XDA set of standard predicates to select *hypotheses* if a given location latitude falls within the interval (45 degrees, 46.2 degrees). The second query uses the existential traversal – hypotheses will be selected if there exists at least one classification type equal to CAR.

Standard Predicates:
SELECT (MHT, Hypothesis_set, hypotheses)
WHERE (location/latitude, greater_than, 45.0)
WHERE (location/latitude, less_than, 46.2)

Existential Traversal:
SELECT (MHT, Hypothesis_set, hypotheses)
WHERE (classifications/?/type, equals, CAR)

User-defined Predicate:
SELECT(MHT, Hypothesis_set, hypotheses)
WHERE(classifications/?, class_of_type, [type,CAR], [belief, 0.7])

Figure 10. Sample queries for target hypothesis ontology

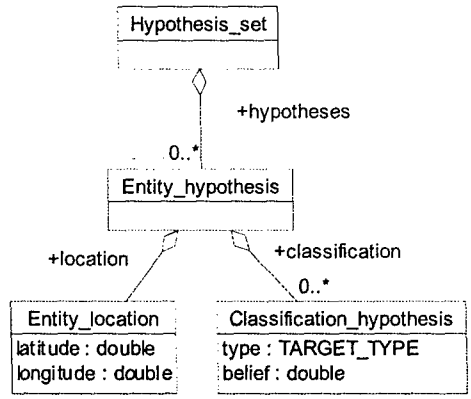

Figure 11. A simple target hypothesis ontology

3.2 Ontology-based visualization tools

The XDA ontology mechanism has been used in integrating visualization tools with heterogeneous data sources. A simple visualization ontology for displaying tracks in geo-spatial coordinates is given in Figure 12. This allows us to visualize products from all fusion components that publish generic visualization objects. This allows loose coupling between the visualization tool and the fusion system since the visualization tool only needs to subscribe for products that are generic visualization objects. An example of a Java visualization tool that uses the OpenMap toolkit is shown in Figure 13.

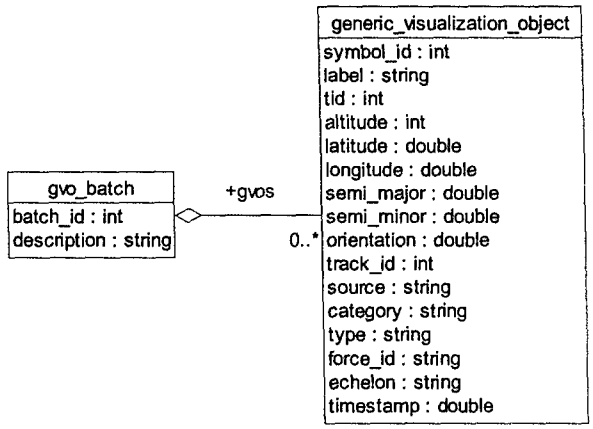

Figure 12. Generic Visualization Ontology

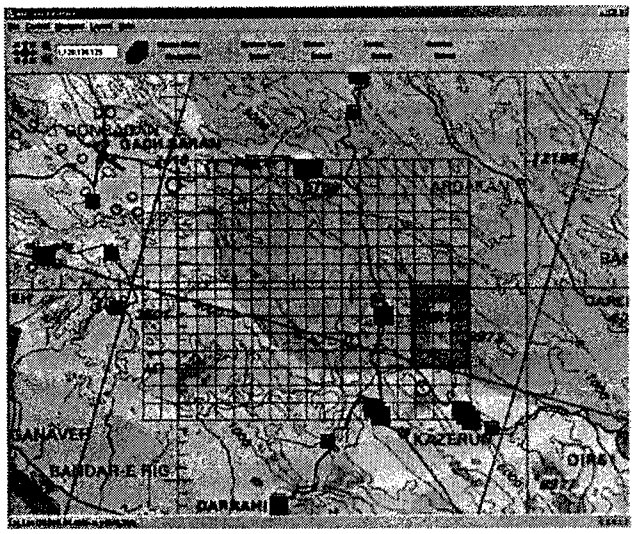

Figure 13. Ontology based visualization

3.3 Ontology-based information pedigree

The pedigree of a given piece of information is a historical record of all information and processing that participated in generating the information. A more detailed description of XDA's approach to information pedigree appears in [4]. Pedigree is an essential element of distributed processing, for the simple reason that data type alone is often not adequate to ensure that a given process can ingest the data. For example a correlator may consume a certain type of sensor feed but will not operate correctly unless the data has been pre-processed by a certain data conditioner. Although pedigree by definition consists of the historical antecedents of a given piece of information, it can also be used for forward tracing that is discovering the consequences of a given piece of information, by querying the pedigree of future information for references to the current piece of information. In the area of information fusion, pedigree is used to detect, and if possible, correct double counting of prior information.

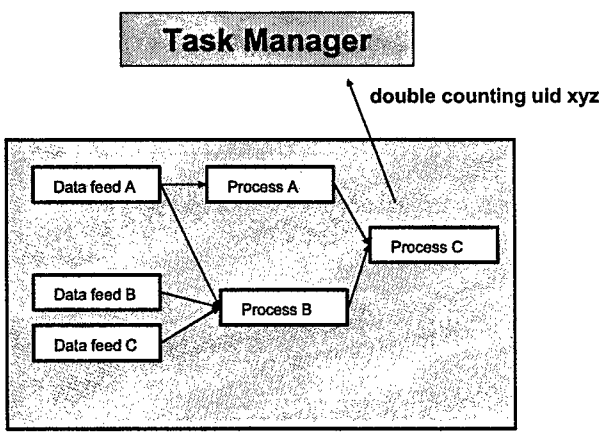

Figure 14. Detecting double counting of evidence

Double counting occurs when the same piece of original information influences downstream processing through multiple processing paths, and it is generally undesirable because it can result in artificially high confidence assessments. Detecting double counting is relatively straight forward given pedigree capability, it is not always possible to easily mitigate its effects.

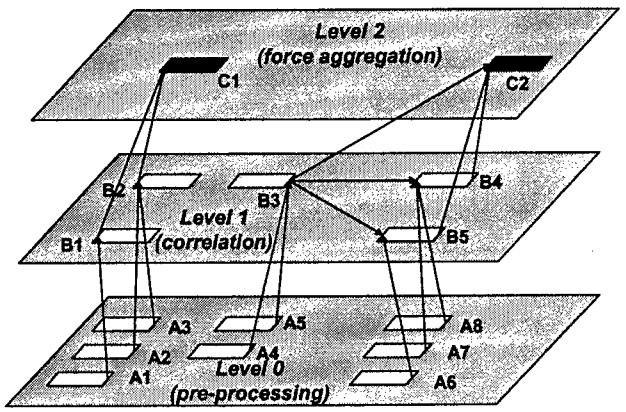

Figure 15. Pedigree can trace the origin as well as the consequences of information

Although information pedigree is essential in distributed systems, pedigree capability is either limited or missing in current systems. In the past, distributed systems have been proprietary, stove-pipe designs with *ad hoc* pedigree functionality at best. Furthermore, there has been a concern that full pedigree capability may adversely affect system performance due to the large amount of additional data needed to capture pedigree information. As the processing streams become longer and more complex, and as the number of collaborating applications increases, the amount and complexity of pedigree information increases, thus straining system resources including storage space, bandwidth and CPU cycles. Utilizing pedigree information involves the application of potentially CPU intensive algorithms. Any information pedigree implementation must address these issues of scalability and runtime performance. We are using XDA's ontology mechanism for pedigree information representation, as well as pedigree storage and retrieval.

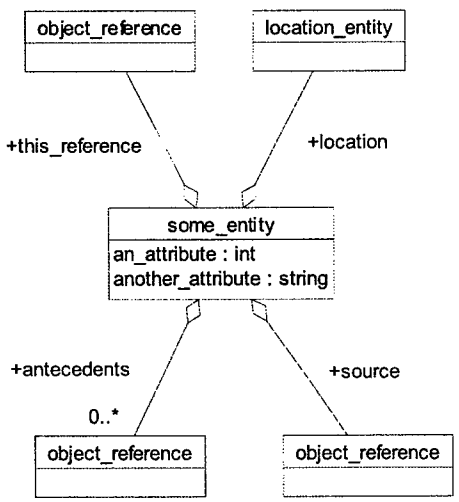

Figure 16. Pedigree-enabled, persistence capable entity

Having an efficient pedigree representation is fundamental to achieving adequate performance. One approach to pedigree representation might be to maintain a separate data structure that contains all the pedigree relationships. This approach has some disadvantages; the pedigree structure can grow very large and must be constantly updated at runtime resulting in database access bottlenecks and reduced performance. Our approach is to avoid maintaining this centralized pedigree repository. Instead, we provide each pedigree-enabled object with just enough additional information to compute its immediate ancestors or antecedents. Consequently, pedigree information is no longer stored as an aggregate whole; it is dynamically computed rather than statically retrieved. However, it is important that adding pedigree capability to an existing entity be as easy and as non-intrusive as possible. Our approach is based on a special attribute called *antecedents* which is a vector of object references to the immediate ancestors of a particular instance as shown in Figure 16.

Presence of the special attribute *this_reference* indicates that an entity is persistence capable. Traversing the information pedigree amounts to traversing through the *antecedents* vectors. In general, an entity instance should be persisted if it contributes to the pedigree of another entity since traversal through an object reference to a transient object can result in an exception. Figure 16. displays a persistence-capable, pedigree-enabled entity that also contains a reference to the source or process that created it.

Once an entity is pedigree enabled any entity derived from it is also pedigree enabled, since it inherits the *antecedents* attribute. Furthermore, pedigreed objects can exist at arbitrary positions in the containment hierarchy of a given object.

4 Conclusion

XDA enables the construction of distributed applications that share a common, object-based domain definition. Use of the ontology mechanism assures loose coupling between the components by providing high-level definitions of the information exchanged between the components. This approach permits the creation of systems that are capable of evolving, with modifications being incrementally deployed as needed. Such modifications can be made at the domain ontology level without affecting the lower-level transport and database schema. Since the ontology definition of the problem domain is no longer coupled with the database and transport schema, significant optimizations become possible in order to assure that such a system can meet operational performance requirements.

Acknowledgments

This work was sponsored by the Air Force Research Laboratory (contract numbers 59272DS102, F30602-98-C0292, and F30602-01-C0045).

References

[1] B. Krikeles, A. Lusignan, E. Starczewski, *A Framework for Distributed Data Fusion*, 2001 MSS National Symposium on Sensor and Data Fusion, San Diego, June 2001.

[2] B. Krikeles, A. Lusignan, E. Starczewski, *Object-based Persistence for Distributed Fusion*, 2002 MSS National Symposium on Sensor and Data Fusion, San Diego, June 2002.

[3] B. Liskov, J. Guttag, *Abstraction and Specification in Program Development*, New York, McGraw Hill, 1986.

[4] A. Chao, B. Krikeles, A. Lusignan, E. Starczewski, *Information Pedigree for Distributed Data Fusion,* 2003 MSS National Symposium on Sensor and Data Fusion, San Diego, June 2003.