CHAOS: An Active Security Mediation System

David Liu[1], Kincho Law[2], and Gio Wiederhold[1]

[1] Electrical Engineering Department, Stanford University, Stanford, CA
davidliu@stanford.edu
[2] Civil Engineering Department, Stanford University, Stanford, CA
law@cive.stanford.edu
[3] Computer Science Department, Stanford University, Stanford, CA
gio@db.stanford.edu

Abstract. With the emergence of the Internet, collaborative computing has become more feasible than ever. Organizations can share valuable information among each other. However, certain users should only access certain portions of source data. The CHAOS (Configurable Heterogeneous Active Object System) project addresses security issues that arise when information is shared among collaborating enterprises. It provides a framework for integrating security policy specification with source data maintenance. In CHAOS, security policies are incorporated into the data objects as active nodes to form active objects. When active objects are queried, their active nodes are dynamically loaded by the active security mediator and executed. The active nodes, based on the security policy incorporated, can locate and operate on all the elements within the active object, modifying the content as well as the structure of the object. A set of API's is provided to construct more complex security policies, which can be tailored for different enterprise settings. This model moves the responsibility of security to the source data provider, rather than through a central authority. The design provides enterprises with a flexible mechanism to protect sensitive information in a collaborative computing environment.

1 Introduction

1.1 Security in Collaborative Systems

The emergence of Internet has greatly extended the scope of collaborative computing. Businesses share information to shorten their product development time; hospitals share information to provide better care to their patients. However, collaborations pose extensive security problems. In fact, protecting proprietary data from unauthorized access is recognized as one of the most significant barriers to collaborative computing [HSRM96].

Software engineers have attempted to apply traditional security approaches to their specific collaborative computing paradigm. Encryption, firewalls, and passwords are used for secure transmission and storage of information [Den83]. User access rights are used in file systems to protect directories and files from unauthorized accesses [GgS91]. These systems rely on domain access control for the security of their data and focus on protecting systems from adversaries. However, they do not properly address the security issues in collaborative computing environments, where information needs to be selectively shared among different domains [JST95]. The following characteristics can be observed in a collaborative computing environment:

1. There is no clear enemy. Users access parts of the information sources. Unless information sources can be broken into small autonomous units, firewalls and passwords cannot provide the functionality needed. If the data sources are finely partitioned, their management becomes complex and difficult.
2. Typically, the information stored in an organization is not organized according to the needs of external accesses. It is in rare cases that security requirements can be properly aligned with organizational needs. For example, medical records are created and organized according to the patients in a hospital rather than according to doctors and staff on whom security clearance needs to be placed.
3. It is impossible to rigorously classify the data by potential recipients. For instance, a medical record on a cardiac patient can include notations that would reveal a diagnosis of HIV, so that this record should be withheld from cardiology researchers. A product specification may include cost of the components provided by suppliers, a competitive advantage that should be withheld from customers.

1.2 Security Mediator

To cope with security issues in collaborative computing environments, security mediators are introduced. Mediators [WG97] are intelligent middleware that sit between information system clients and sources. They perform functions such as integrating domain-specific data from multiple sources, reducing data to an appropriate level and restructuring the results into object-oriented structures. The mediators that are applied to security management are called *security mediators* [WBSQ96b]. An example of a security mediation system is the TIHI project [WBSQ96a], in which a rule system is used to automate the process of controlling access and release of information. Applicable rules are combined to form security policies, which are enforced by the mediator for every user. Results are released only if their contents pass all tests. This model (**Figure 1**) formalizes the role of a mediation layer, which has the responsibility and the authority to assure that no inappropriate information leaves an enterprise domain.

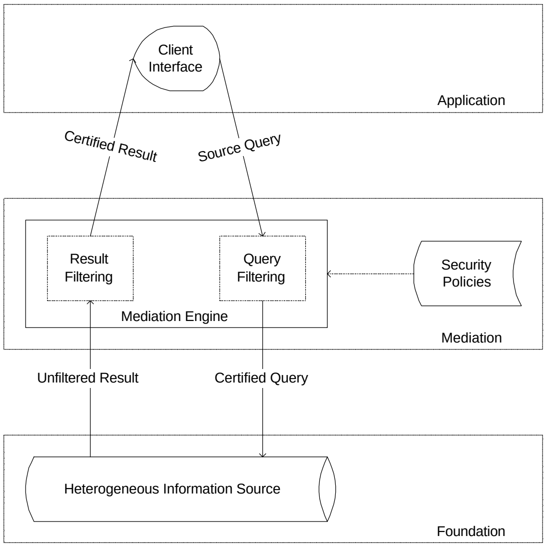


Figure 1: Static Security Mediation

Security rules act like meta-data in a database. They are predefined by the security expert for the system and are applied to data items that are returned from the queries. Since all rules are statically specified and checked, we call this type of system *static security mediation* system. In such systems, there is a security officer whose responsibility is to implement and control the enterprise policies set for the security mediator. Databases and files within the domain provide services and meta-data to help the activities of the security mediator.

While static security mediation addresses a broad range of security issues in collaborative computing, it suffers certain shortcomings that motivate the proposed approach to move security policies from the mediation layer to the foundation layer and to give more flexibility in specifying security policies.

First of all, in many scenarios, it is natural to have the information source set and manage its own security policy. A heterogeneous information system may organize its source data as information islands, and each island is maintained distinctively from the others. This organization is becoming more pervasive for Internet services. We observe that source data maintenance and security policy specification are tightly related in these situations. When source data get updated, especially when their data structure changes, the related security policies may need to be modified accordingly.

Secondly, it is difficult to design a rule base security mediator that fits a broad range of heterogeneous information systems. Enterprise security policies are specified in terms of the primitive rules predefined for the static mediation system, making it difficult to develop a comprehensive set of rules that can be effectively combined to satisfy a very broad range of security needs.

Generally, rules are best applied to relational databases since they are defined on table schemas. In the case of unstructured data that lack a predefined schema, rules are difficult to apply. Furthermore, acting as meta-data in a database, rules act on tables. They are most suited to filter out rows of data entries, but lack the capability to prune the structure of the result entries to allow partial access to the data. Traditional view based access control system [GW76] could be used to amend this deficiency. Separate views can be constructed for each partial structure while appropriate access rights can be assigned to each view. However, this approach is similar to that of domain access control. Managing views and maintaining their secret labels become very complex as the system grows [WBSQ96b].

1.3 Active Security Mediation

We propose a solution to these problems in CHAOS. We define a special type of objects, *active objects*, which incorporate security policies into data objects as *active nodes*. Rather than treating rules as meta-data acting on tables, we enforce security by invoking functions contained in active nodes that act on data objects. The design of CHAOS is schematically shown in **Figure 2**.

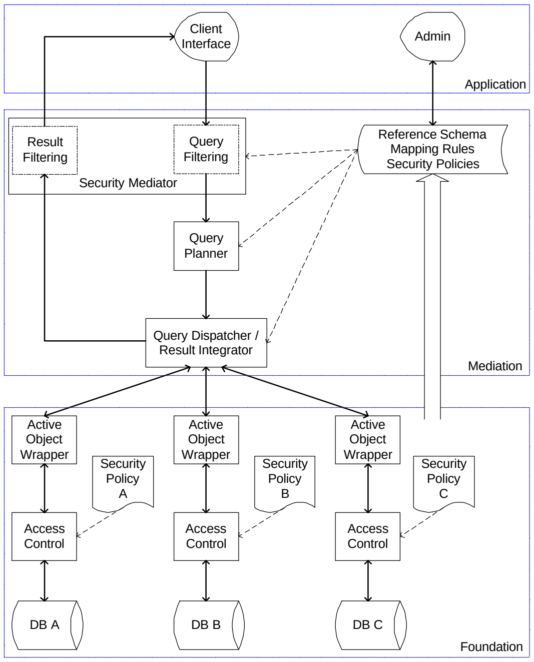


Figure 2: CHAOS Active Security Mediation

In CHAOS, each information source is treated as an information island that has its own access control policies. An incoming client query request is first checked by a Query Filtering module, where unauthorized request to the heterogeneous system are denied. The Query Planner and Query Dispatcher modules are in place to decompose a client query into source queries that individual heterogeneous sources can answer.

The methods of query transformation belong to a different scope of schema integration, hence are not discussed in detail here. Upon receiving query requests from the mediation layer, the foundation layer sources fetch the query results, wrap them as active objects, and pass the active objects onto the mediation layer. The Result Filtering module will interpret encapsulated active nodes and translate active objects into regular data objects before passing them onto the client.

In the TIHI model [WBSQ96a], it is assumed that the people controlling the sources do not care much about security. That is true for many medical doctors, who willingly share data and do not realize how far the data might spread and embarrass the patients. When private information gets leaked, it is the institution, as the holder of the data, who assumes the responsibility. In the CHAOS model the assumption is that the owners of the data care about the security of the data, often for competitive business reasons, sometimes perhaps even being competitive within an institution. This model fits those institutions that delegate much authority to enterprise units.

By incorporating active nodes into data objects, we provide a tight integration between security policy specification and source data maintenance. Each data object has a clear view of all policies that are applicable to it. Furthermore, security policies can be applied to individual data objects, providing a fine grain of control. We use Java as the active node specification language, giving greater expressive power to the security system. For the ease of system configuration and maintenance, we provide an extendible set of API's that allow more complex policies to be composed. At the same time, unlike static security mediation system where policies are solely based on primitive rules, CHAOS does not place any restriction on whether active nodes use API's to manipulate their objects.

2 CHAOS System Design

2.1 Active Object

Objects are used as the basic data model to describe source data in the CHAOS. Most clients are best served by information in object-oriented form that may integrate multiple heterogeneous sources [PAGM96]. Specifically, in CHAOS, data are represented in XML [W3C98]. Such choice is made because of XML's nature of extensibility, structure, and validation as a language. However, the concept and our system design can be easily extended to other data models. In subsequent section we show a sample application of the CHAOS system architecture that uses a relational database as the source data repository.

XML is a meta-markup language that consists of a set of rules for creating semantic tags used to describe data. An XML element is made up of a start tag, an end tag, and content in between. The start and end tags describe the content within the tags, which is considered the value of the element. In addition to tags and values, attributes are provided to annotate elements. For a more comprehensive description of XML, see [W3C98]. In essence, XML provides the mechanism to describe a hierarchy of elements that forms the object.

Active object is a special type of XML object. In active objects, two types of elements are defined: data elements and *active elements*. A data element, like any regular XML elements, describes the content of an object; an active element, on the other hand, no longer describes the content of an object but rather contains the name of an *active node* that operates on the object and generates the content. We use attributes to identify active elements by setting their *active-node* attribute to *true*.

2.2 Active Element

Each active element contains one *active node*, a Java class that will be interpreted by the mediator runtime environment. Java [SUNb] is chosen as the function description language because of Java's support for portability, its flexibility as a high-level language, and its support of dynamic linking/loading, multi-threading and standard libraries.

All active nodes are derived classes of *ActiveNode* (See Appendix A), and they overload the *execute* function to provide specific functionality. The *execute* function takes three parameters: the current active element handle, the root element handle, and the client environment information. The mediator runtime environment fills in these three parameters when the mediator loads the active nodes during the runtime.

Java Project X [SUNa], a Java based XML service library package, is preloaded into the CHAOS security mediator runtime environment. The package provides core XML capabilities including a fast XML parser with optional validation and an in-memory object model tree that supports the W3C DOM Level 1 recommendation [W3C]. Using the API's provided by the package, we can parse XML documents, query elements in an XML object, and modify the content and structure of the object.

In order for active nodes to interact with data elements in an active object, a mechanism is needed to locate all elements. We employ the concept of label path [GW76] from the LORE [MAG+97] project and define *tag path*:

Definition: A tag path of an element e_0 is a sequence of one or more dot-separated tags, $t_1(s_1).t_2(s_2)\ldots t_n(s_n)$, such that we can traverse a path of n elements $e_1,e_2,\ldots,e_n$ from e_0 where node e_i is the child of e_{i-1} and is the s_i- th child that has the tag t_i. In case where s_i is not specified, its default value is 1.

With the tag path definition, active nodes can uniquely locate an element *e* by specifying the root element of the object and a tag path that traverse from the root element to *e*. All elements within an active object can be reached and manipulated by the active nodes that are contained in the object. In the cases where multiple active objects are to be manipulated by a common active node, the active objects can be combined together to form a larger object such that a common root element can be provided to the *execute* function.

The CHAOS system provides a set of *ActiveNode* API's. Elements within an active object can be queried, structure of an active object can be altered, and statistical information about an active object or an element can be generated. Based on these API's, more complex functionality can be constructed. As opposed to static mediation system, where policies are constructed based on primitive rules, *ActiveNode* API's place no limitation on how the policies can be constructed. The API's are provided for mere convenience rather than for restriction.

With multiple active nodes in an active object, the order in which they are executed may affect the final mediated result. CHAOS adopts a depth first ordering approach in loading and interpreting active elements within an active object.

2.3 Security Mediator

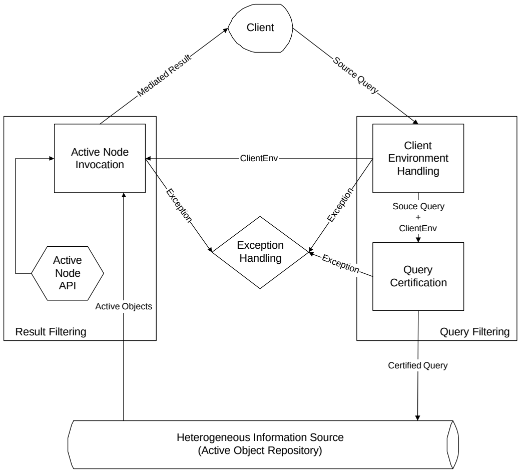


Figure 3: Active Security Mediator Architecture

Security mediator is the component in CHAOS where client source query is parsed and certified, active objects are queried and interpreted, and mediated results are returned. As shown in **Figure 3**, a security mediator is composed of two main modules: Query Filtering and Result Filtering. In addition, an exception-handling module is inserted in case abnormal system behavior occurs.

1. *Query Filtering Module*: A Query Filtering module deals with parsing and certifying incoming query request to the heterogeneous source. The Client Environment Handling component customizes the active security runtime environment depending on specific client. Client environment information is put into a system defined *ClientEnv* object, which will be passed onto Query Certification and Active Node Invocation Components. Similar to the TIHI system [WBSQ96a][WBSQ96b], the mediator processes the incoming query and checks for its validity. The Query Certification components can look up a static table of rules. Based on the incoming query request and client environment information, it restructures and forwards the query to the underlying heterogeneous information source.
2. *Result Filtering Module*: The core of the Result Filtering module is the Active Node Invocation component. For all the incoming active objects, the component identifies the active nodes in an object and dynamically loads in the appropriate

encapsulated active nodes. Active nodes are invoked by calling their *execute* function with parameters assembled by the mediator. Active nodes will operate on the active objects that contain them. The resulted objects will be forwarded to the client. An Active Node API library is provided to facilitate security policy construction. As indicated in the previous section, all active nodes are derived from *ActiveNode* class. Useful functions and class definitions are put in the *ActiveNode* class. They can be accessed as API's through *ActiveNode*'s method interface. The library is preloaded into the mediator for dynamic linking and invocation by active nodes.

3. *Exception Handling*: It is critical for the system to have a comprehensive exception handling policy. Our current implementation prohibits any results from getting through the mediator in the case of exception. In addition, the conditions are logged for future maintenance.

3 System Implementation

In this section, we describe our implementation of a sample business inventory system using CHAOS system architecture. We study the quality of our design by comparing it with other alternatives that can be chosen to achieve the same objectives.

In order to better compete in the marketplace, businesses have the need to streamline their procurement and distribution processes which requires integration of all relevant data. Our example considers a PC company, for whom it is important to deliver the product design information and pricing information to its distributor in a timely and convenient manner. At the same time, it is critical to protect its cost structure from the competitors.

The source data are originally stored in ORACLE, a relational database, on top of which a CHAOS system is built to provide integrated product information. The schemas are shown in **Figure 4**.

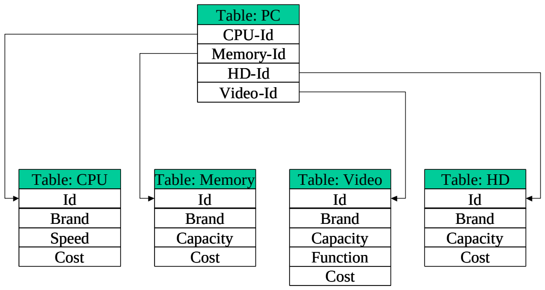
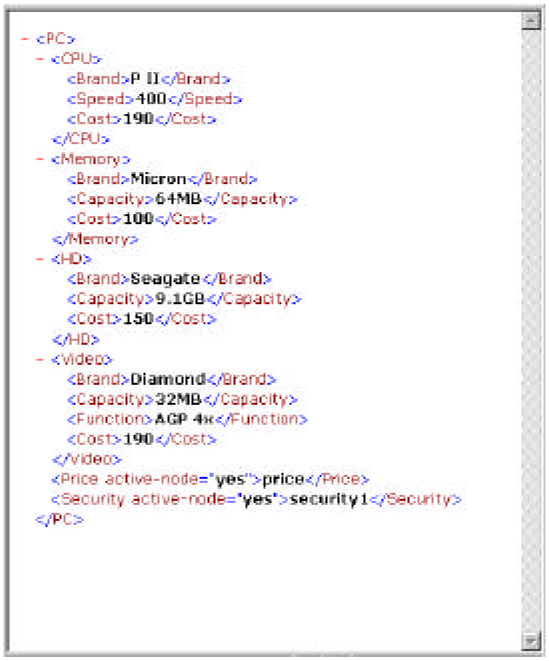


Figure 4: Relational Schemas for an Inventory Database

In a heterogeneous information system, no particular data model can be assumed for the original information source. Using XML, however, we can pack heterogeneous data into uniform logical objects. Integrating heterogeneous information is a research issue that is addressed in [PAGM96][WG97]. Necessary mediators are added to perform the data conversion. Therefore, we treat the heterogeneous information source in CHAOS as an active object repository. All source data are converted into active objects before they are exchanged between the foundation layer and the mediation layer of the information system. Necessary mediators are added to perform the data conversion.

For our example, relational data stored in ORACLE database are converted into active objects. Together with regular data elements, active elements are assembled to form active objects. **Figure 5** shows a sample active object that is assembled by a source mediator that queries the relational information source.



```
- <PC>
  - <CPU>
      <Brand>P II</Brand>
      <Speed>400</Speed>
      <Cost>190</Cost>
    </CPU>
  - <Memory>
      <Brand>Micron</Brand>
      <Capacity>64MB</Capacity>
      <Cost>100</Cost>
    </Memory>
  - <HD>
      <Brand>Seagate</Brand>
      <Capacity>9.1GB</Capacity>
      <Cost>150</Cost>
    </HD>
  - <Video>
      <Brand>Diamond</Brand>
      <Capacity>32MB</Capacity>
      <Function>AGP 4x</Function>
      <Cost>190</Cost>
    </Video>
    <Price active-node="yes">price</Price>
    <Security active-node="yes">security1</Security>
  </PC>
```

Figure 5: Sample Active Object

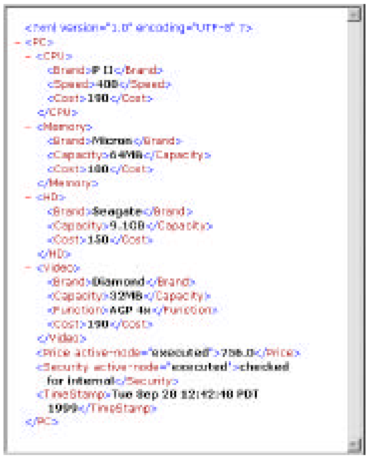
There are two active nodes contained in the active object: *price* and *security1*, which will be dynamically loaded and interpreted by the security mediator when the object is passed through the security mediator. The price information will be generated by the active node *price*, and security will be enforced by the active node *security1*. Both *price* and *security1* are written in Java (See Appendix B) and are based on the *ActiveNode* API's provided by the system.

The active node *price* sums up the cost of all elements and then multiply the cost by a factor (here specified as *1.2*). Effectively, active nodes provide a simple mechanism to specify derived data, thereby maintaining data dependency among various components in an object. To achieve the same objective with traditional database system designs, few other alternatives can be considered, each with certain drawbacks. A database update program can be run on top of a relational database to maintain the data dependency. However it is extremely difficult if not impossible to determine the optimal update frequency. As an alternative, every client application can embed a procedure that updates the database on queries. Obviously, this approach is a software engineering nightmare. Active database systems [Day88] could also be considered to address this issue. These systems integrate production rules facility into the conventional database systems. Production rules can be considered as the bonding between the data and the functions in a database system. However, data are required to be migrated into a single active database, a very difficult process for heterogeneous data sources. With active objects, such migration is not needed.

The responsibility of the active node *security1* for the active object shown in **Figure 5** is to filter the information and reshape the structure of the active object based on the client that makes the query. For different clients, certain information needs to be withheld. Only internal users are granted the access to the cost structure of any product. Specified in the active node *security1*, if the client is not an *internal* user, the *Cost* elements of all components will be trimmed, hence withholding the confidential information.

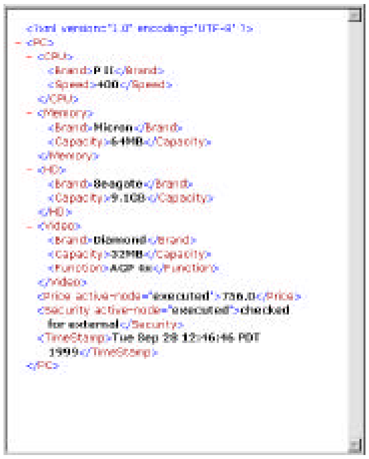
Comparing to a view based access control system, an alternative to address the partial data access issue, CHAOS is cleaner and more flexible. In a view based access control system, different views need to be specified, each with a different schema. For two different user groups, that may not seem to be a great challenge. However, maintenance of views and their secret labels can quickly become a significant management problem when the complexity of the security needs grows. For example, if we want to change the security policy to allow each procurement department access to the cost information of their own components but not the cost information of the other components, four more views need to be specified and maintained. Whereas in CHAOS, no separate views need to be constructed. The information source can specify the policy for partial access to its content by adding few more clauses in the active node *security1*.

The results for different client queries are shown in **Figure 6** and : internal Result **Figure 7**. The internal result is generated by a query submitted by a client who belongs to the *internal* clique. All component information and pricing information are returned in the result object. In addition, the active node *security1* adds a time-stamp element to the object. The external result is generated for an *external* client. Comparing to the internal result, external result does not contain any component cost information, which is pruned by the active node.



```
  <?xml version="1.0" encoding="UTF-8" ?>
- <PC>
  - <CPU>
      <Brand>P II</Brand>
      <Speed>400</Speed>
      <Cost>190</Cost>
    </CPU>
  - <Memory>
      <Brand>Micron</Brand>
      <Capacity>64MB</Capacity>
      <Cost>100</Cost>
    </Memory>
  - <HD>
      <Brand>Seagate</Brand>
      <Capacity>9.1GB</Capacity>
      <Cost>150</Cost>
    </HD>
  - <Video>
      <Brand>Diamond</Brand>
      <Capacity>32MB</Capacity>
      <Function>AGP 4x</Function>
      <Cost>190</Cost>
    </Video>
    <Price active-node="executed">750.0</Price>
    <Security active-node="executed">checked
      for internal</Security>
    <TimeStamp>Tue Sep 28 12:42:40 PDT
      1999</TimeStamp>
  </PC>
```

Figure 6: internal Result



```
  <?xml version="1.0" encoding="UTF-8" ?>
- <PC>
  - <CPU>
      <Brand>P II</Brand>
      <Speed>400</Speed>
    </CPU>
  - <Memory>
      <Brand>Micron</Brand>
      <Capacity>64MB</Capacity>
    </Memory>
  - <HD>
      <Brand>Seagate</Brand>
      <Capacity>9.1GB</Capacity>
    </HD>
  - <Video>
      <Brand>Diamond</Brand>
      <Capacity>32MB</Capacity>
      <Function>AGP 4x</Function>
    </Video>
    <Price active-node="executed">750.0</Price>
    <Security active-node="executed">checked
      for external</Security>
    <TimeStamp>Tue Sep 28 12:46:46 PDT
      1999</TimeStamp>
  </PC>
```

Figure 7: external Result

4 Conclusion

The CHAOS system provides a framework for integrating security policy maintenance with data source maintenance. Active nodes are incorporated into the data objects of which they control the security policy. They can locate and operate on all elements within the active object, modifying the content as well as the structure of the object. This approach moves the responsibility for security to the source provider, rather than through a central authority.

Unlike rule based security systems, policies in CHAOS are specified in a general programming language. The system does not need to rely on an initial set of primitive rules to be functional. We provide a set of API's that can be used to construct more complex and powerful policies. These API's are provided for mere convenience and can be expanded. This approach offers much greater flexibility.

Similar to most security measures, the active security mediation does not offer 100% guarantee. It is restrained by the quality of the system design and the implementation of the policies. However, it provides a clear, simple and powerful mechanism to serve enterprise policies effectively.

5 Acknowledgement

This work is partly supported by the Center for Integrated Facility Engineering at Stanford University, by the National Science Foundation under grant ECS-94-22688, and by DARPA/Rome Laboratory under contract F30602-96-C-0337. The authors would also like to acknowledge a “Technology for Education 2000” equipment grant from Intel Corporation in support of the research.

References

[Day88] U. Dayal. Active database management systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, 1988.

[Den83] Dorothy E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, MA, 1983.

[GgS91] Simson Garfinkel and Gene Spafford. *Practical Unix Security*. O'Reilly and Associates, Inc., 1991.

[GW76] P. P. Griffiths and B. W. Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems*, 1(3):243-255, Sept. 1976.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. *VLDB Conference*, 1997.

[HSRM96] Martin Hardwick, David L. Spooner, Tom Rando, and K.C. Morris. Sharing manufacturing information in virtual enterprises. *Comm. ACM*, 39(2):46-54, Feb. 1996.

[JST95] D. Randolph Johnson, Fay F. Sayjdari, and John P. Van Tassell. Missi security policy: A formal approach. Technical Report R2SPO-TR001-95, National Security Agency Central Service, July 1995.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54-66, Sept. 1997.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. *VLDB Conference*, 1996.

[SUNa] SUN Microsystems Inc. Java Project X. 1999. http://developer.java.sun.com/developer/products/xml.

[SUNb] SUN Microsystems Inc. The source for java technology. 1995-1999. http://www.javasoft.com.

[W3C] W3C. Document object model. 1998. http://www.w3.org/DOM.

[W3C98] W3C. Extensible markup language. 1998. http://www.w3.org/XML/.

[WBSQ96a] Gio Wiederhold, Michel Bilello, Vatsala Sarathy, and XiaoLei Qian. Protecting collaboration. In *Proceedings of the NISSC'96 National Information Systems Security Conference*, pages 561-569, Oct. 1996.

[WBSQ96b] Gio Wiederhold, Michel Bilello, Vatsala Sarathy, and XiaoLei Qian. A security mediator for health care information. In *Proceedings of the 1996 AMIA Conference*, pages 120-124, Oct. 1996.

[WG97] Gio Wiederhold and Michael Genesereth. The conceptual basis for mediation services. *IEEE Expert, Intelligent Systems and their Applications*, 12(5), Oct. 1997.

Appendix

A.1 Sample ActiveNode API

```
public class ActiveNode
{
    /*
     * Entry point, needs to be overloaded
     */
    public String execute(Element current, Element root,
                          ClientEnv env);

    /*
     * Query elements within an active object
     */
    protected Node getNode(Element root, String path);
    protected String getString(Element root, String path);
    protected int getInt(Element root, String path);

    /*
     * Manipulate structure of an active object
     */
    protected Node removeNode(Element root, String path);
    protected void removeAllNode(Element root, String tag);
    protected void appendNode(Element root, String path,
                              Node child);
    protected void appendNode(Element root, String path,
                              Node child);

    /*
     * Miscellaneous statistical functions
     */
    protected int sumAllNodes(Element root, String tag);
    protected int getNumChildren(Element root, String tag);

    /*
     * Utility functions
     */
    protected void initLog(boolean onoff);
    protected void log(String msg);
}
```

A.2 Sample Active Nodes

Active Node *price*

```
public class price extends ActiveNode
{
    public String execute(Element current, Element root,
                        ClientEnv env)
    {
        int cost = sumAllNodes(root, "Cost");

        return String.valueOf(1.2 * cost);
    }
}
```

Active Node *security1*

```
public class security1 extends ActiveNode
{
    public String execute(Element current, Element root,
                        ClientEnv env)
    {
        /* Check the clearance of the client */
        if (!env.Clique().equals("internal")) {
            removeAllNodes(root, "Cost");
        }

        /* Add a time stamp to the object.  */
        createTextElement(root,
                           "",
                           "TimeStamp",
                           new Date().toString());

        return "checked for " + env.Clique();
    }
}
```