A Policy-based Approach for Assuring Data Integrity in DBMSs

Hyo-Sang Lim
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907
hslim@cs.purdue.edu

Chenyun Dai
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907
daic@cs.purdue.edu

Elisa Bertino
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907
bertino@cs.purdue.edu

***Abstract*—Data integrity is crucial for collaborative activities where information is shared among multiple organizations to effectively make cooperative and mission-critical decisions. Assuring data integrity is particularly challenging in the presence of frequent data modifications by collaborative parties, especially for large-scale collaborations. However, data integrity is difficult to grasp with a single concept or a single model since the definition can vary depending on the goals and requirements of the collaboration. To address this multi-faced feature of data integrity, we propose a policy-based approach by which one can specify data integrity policies according to the requirements of collaborations and enforce the policies on DBMSs, an essential software component for large-scale collaboration activities. We first introduce our integrity policy language, which provides comprehensive framework for specifying and enforcing integrity policies based on access control, data validation, and metadata management functions. Next, to make our policy language work with existing off-the-shelf DBMSs, we present an integration strategy which we call language level integration (LLI). The LLI strategy enforces integrity policies by automatically translating high-level integrity policies, expressed in our policy language, onto low-level database operations. Compared to alternative approaches, the LLI strategy can be easily implemented since it does not require modifications to the source code of the DBMS or to the code of the applications running on top of the DBMS. Also, with the LLI strategy, the policies cannot be bypassed regardless of which database interface is used by the applications since the policies are implemented by DBMS functions and objects, and then, automatically enforced by the DBMS itself. We then present a software architecture of implementing the integrity policy language with the LLI strategy for a real DBMS (Oracle) and show that our strategy can easily implement well-known data integrity models.**

I. INTRODUCTION

Data integrity is crucial for collaborative activities where organizations share information within and across the organizations so that analysts and decision makers can analyze the data, mine the data, and make cooperative and mission-critical decisions effectively. The problem of data integrity becomes particularly challenging for large-scale collaborations in which parties make frequent data modifications. Without integrity, collaborations among multiple organizations cannot be successful since the usefulness of data becomes diminished as any information extracted from them cannot be trusted with sufficient confidence.

Despite the significance of the problem, theoretical/technical solutions available today for integrity are still very limited. A key difficulty comes from the fact that the concept of integrity is difficult to grasp with a precise definition. The most widely accepted definition of integrity is perhaps *prevention of unauthorized and improper data modification* [1]. This definition also seems to coincide with the primary goal of Clark and Wilson's approach, "preventing fraud and error" [5]. Another well-known interpretation of integrity concerns the *quality and trustworthiness of data* [3]. Inspection of mechanisms provided by database management systems (DBMSs) suggests yet another view of integrity. Many commercial DBMS today enable users to express a variety of conditions, often referred to as integrity constraints, that data must satisfy [6]. Such constraints are used mainly for *data consistency and correctness*. This multi-faceted concept of integrity makes it challenging to adequately address integrity, as different definitions require different approaches. For instance, Clark and Wilson address the issue of improper data modification by enforcing well-formed transactions and "separation of duty" [5], whereas the Biba's integrity model prevents possible data corruption by limiting information flow among data objects [2]. On the other hand, many current DBMSs ensure data consistency by enforcing various constraints, such as key, referential, domain, and entity constraints [6].

To address the problem of high-assurance data integrity, we propose a policy-based approach by which one can specify high-level data integrity policies according to the requirements of collaborations and enforce the policies on DBMSs. Our policy-based approach is based on the following elements:

- A reference architecture for integrity management which supports both integrity-related *access control* and *data validation*.
- The notion of *integrity metadata template* for specifying metadata information relevant for integrity.
- A flexible *integrity control policy language* able to address several integrity requirements.
- An approach for automatically enforcing integrity policies and metadata templates on top of existing DBMS.

The last element is a crucial component of our solution in that it makes possible to deploy our high-assurance integrity

solution as a layer on top of current DBMS. Because such layer implements our integrity policy language, our solution achieves the goal of providing an integrity solution which does not require modifications to the source code of the DBMS and at the same time does not require modification to the code of the applications running on top of the DBMS.

In this paper, we first introduce our integrity policy language which is developed as part of our previous work [4] and extended for use in DBMSs. We then present integration strategies to make our policy language work with existing off-the-shelf DBMSs. We consider three alternative strategies, namely integration at *source code* level, at *application level*, and at *language* level. Based on a comprehensive analysis, we identify the language level integration (LLI) strategy as the best since it is not only the easiest to implement but it also achieves a robust enforcement of policies. The LLI strategy can be easily implemented on top of current DBMSs since it does not require modifications to the source code. Since the LLI strategy utilizes DBMS built-in facilities such as triggers, it also provides a robust enforcement of integrity policies regardless of which database interface (e.g., web-database gateway, call level interface (CLI), ODBC/JDBC driver) is used by the applications running on top of the DBMS. Finally, we provide a software architecture for implementing the integrity policy language according to the LLI strategy and discuss in details our translation approach for a specific DBMS, that is, Oracle. We also show that our strategy can easily implement well-known data integrity models such as Biba's model.

The remainder of this paper is organized as follows. Section II introduces our integrity policy language. Section III introduces the three implementation strategies for the integrity policy language and analyzes the advantages and disadvantages of each strategy. Section IV introduces the architecture implementing the integrity policy language according to the LLI strategy and discusses the automatic translation of integrity policies onto Oracle DBMS. Section V reports the results of the automatic translation, whereas Section VI concludes the paper.

II. Integrity Policy Language

The integrity policy language represents a flexible mechanism to specify which actions the database system should take in order to assure data integrity. In what follows, we first describe a language for specifying metadata for managing integrity policies and then present our integrity policy language which supports both data validation and integrity-related access control. The integrity policy language has been designed based on the relational data model since this model is supported by almost all DBMSs. The language also targets, for the component concerning access control, the role-based access control (RBAC) model since this model is the most popular access control model in many application domains and supported by almost all DBMSs. Therefore, we consider the data objects to be data items (i.e., tuples) in tables (i.e., relations), and the subjects, that is, the active entities accessing and manipulating the data objects, to be users belonging to roles in a RBAC system.

A. Metadata specification language

In this paper, we refer to information based on which data integrity is determined as *metadata*. Such information can vary depending on the type of data and/or the requirements of collaborations. For instance, one can evaluate the integrity of a particular data item based on the role which created the data, the source from which the data is obtained, or the value of some other data items that are related to the data. To allow the application to specify and manage this information, we introduce the notion of *metadata template* which is an abstract data structure for metadata.

Metadata templates are thus the basis of the specification and enforcement of integrity policies and are specified by the metadata specification language. Therefore, metadata templates are essentially pre-defined, specific descriptions (e.g., names and types of attributes) of data, which are relevant for the integrity of objects (i.e., data items). Metadata templates are also defined for the subjects (i.e., users) to describe various attributes of the subjects, which are necessary to make integrity related access control decisions. In our language, metadata templates for objects are defined at the table level and metadata templates for subjects are defined at the role level. This means that every data item in a table or every user belonging to a role is associated with the same metadata templates.

In addition to defining a set of attributes for a table or a role, metadata templates also specify how each defined attribute should be initialized and managed. More specifically, each attribute in a metadata template is associated with a specific method which determines the value of the attribute; that is, an attribute is registered with a default value, a designated function, or a system variable such as $USER or $TIME. Except for the attributes that are registered with default values, the attribute values must be updated only through the registered procedure or a system variable. We note that such controlled management of the metadata attributes is necessary to guarantee the integrity of metadata values.

When a new data item is introduced to a system, an instance of metadata template (i.e., a metadata item) is created for the data item, according to the metadata template specified for the type of the data item. Then this metadata object is associated with the data item throughout its life-cycle; that is, whenever an access to the data item is requested, the metadata item is retrieved and possibly updated according to the related integrity control policies. Similarly, when a user activates a role, a metadata item is instantiated from the metadata template specified for the role. This metadata item is associated with the user and used for managing data integrity until the user deactivates the role. A metadata template is created according to commands specified as Syntactic Rule 1.

We can see from Syntactic Rule 1 that the command for creating metadata templates is similar to that for defining

Syntactic Rule 1: Create Metadata Template

CREATE MD-TEMPLATE *template_name* FOR *target_type* : *target* {
    $attr\_name_1$ $attr\_type_1$: $attribute\_description_1$;
    . . .
    $attr\_name_n$ $attr\_type_n$: $attribute\_description_n$;
}

where:

- *template_name* : the unique identifier for the metadata template.
- *target_type* : table, role. If *target_type* is 'table', the metadata template is created for the relational table whose name is *target*. If *target_type* is 'role', the metadata template is created for the set of users whose role is *target*.
- *target* : the name of a table or a role corresponding to the metadata template.
- $attr\_name_i$: the name of the i-th attribute.
- $attr\_type_i$: the data type of the i-th attribute.
- $attribute\_description_i$: the registered method for the i-th attribute, which may be a specific value, a function, or a system-variable. □

a table in relational DBMSs. However, in addition to the specification for attributes such as names (*attr_name*) and types (*attr_type*), Syntactic Rule 1 specifies which table or role should be associated with the metadata template (*target* and *target_type*) and the methods for initializing the value of each attribute (*attribute_description*).

B. Integrity policy specification language

Like metadata templates, an integrity control policy is specified for a particular table and enforced on every tuple in the table. There are two kinds of policies in our framework: access control policies (ACPs) and data validation policies (DVPs). ACPs are essential for integrity control as modifications to data may have a direct impact on data integrity. The ACPs are also necessary for addressing the issue of undesirable information flow (e.g., [2]) through a series of retrieval and modification operations. Compared to the conventional access control in DBMSs, we note that the purpose of the ACPs is to prevent 'improper' accesses, not 'unauthorized' accesses. That is, the ACPs do not deal with whether or not users have proper privileges to access data, but only deal with whether or not data are properly accessed by authorized users. The other key component of our integrity control policy is represented by the DVPs which govern the continuous process of monitoring and/or enhancing the integrity of data. Compared to the ACPs, a unique characteristic of the DVPs is that they monitor the data independently from accesses or modifications. This kind of autonomous monitoring process is essential when the integrity of data depends on dynamic factors such as time or real-world events. Compared to the ACPs which only consider the data item accessed, the DVPs can verify all data items in a table to enhance overall data integrity. ACPs and DVPs are defined according to Syntactic Rule 2 and 3, respectively.

In Syntactic Rule 2 and 3, the WHEN-clause specifies the particular event that triggers the specified policy. The ACPs are triggered only by access requests (*ac_event*) while the DVPs may be triggered by either access requests or some user-defined events (*event*). The user-defined events include a time event (i.e., the event occurs for each certain time period), a counter event (i.e., event occurs for every certain number

Syntactic Rule 2: Create Access Control Policy (ACP)

CREATE ACP *acp_name* FOR (*table_name*, *role_name*) {
    WHEN *ac_event*;
    IF *condition*;
    THEN *then_decision*: *then_action*;
    ELSE *else_decision*: *else_action*;
}

where:

- *acp_name* : the unique identifier for the ACP.
- *table_name*: the name of the table on which the policy is enforced.
- *role_name*: the name of role who invokes the policy in a certain event.
- *ac_event* : represents an access request {Read, Insert, Update, Delete}.
- *condition* is a set of boolean-expression primitives which may be conjuncted, disjuncted, or negated with the boolean operators $\wedge$, $\vee$, and $\neg$, respectively.
- *then_decision, else_decision* : is an access control decision which is one of {Allow, Deny}.
- *then_action, else_action* : represents an action to be taken as a consequence of the corresponding access control decision. An action is either a procedure invocation or a metadata update. □

Syntactic Rule 3: Create Data Validation Policy (DVP)

CREATE DVP *dvp_name* FOR *table_name*
    WHEN $event_1, \ldots, event_l$;
    IF *validation_procedure*;
    THEN *then_action*;
    ELSE *else_action*;
}

where:

- *dvp_name*: the unique identifier for the DVP.
- *table_name* : the name of the table on which the policy is enforced.
- $event_k$, $k = 1, \ldots, l$, represents either an access request {Read, Insert, Update, Delete} or a user-defined event such as a specific time or a particular situation that triggers the specified validation policy.
- *validation_procedure* is a designated function which validates the data instances of $table\_name$. It returns *true* if the validation succeeds; otherwise, it returns *false*.
- *then_action, else_action* : represents an action to be taken as a consequence of the data validation. An action is either a procedure invocation or a metadata update. □

of new data tuples), and alarms (or signals) from outside the system.

The IF-clause in an ACP contains a condition (*condition*) that checks various metadata attributes in order to determine the integrity of the data. After evaluating the condition, either the THEN-clause or the ELSE-clause is executed. Each THEN-clause and ELSE-clause contains an access control decision (*then_decision*, *else_decision*) which may be either allow or deny, and also a set of actions (*then_action*, *else_action*) that should be taken subsequently. Possible actions include updating metadata attributes or invoking necessary procedures.

The IF-clause in a DVP contains a data validation procedure which returns the result of the data validation. Like in ACPs, each THEN-clause and ELSE-clause in DVPs specify a set of actions that should be taken according to the result of the validation procedure.

C. Running Example

For illustrating how metadata templates and integrity control policies are specified and enforced to address various integrity

requirements, we introduce a simple usage scenario.

1) A Simple Application Scenario: A Financial Company: This application scenario concerns a fictitious financial company, IntegrityEqualsMoney (IEM). The goal of IEM is to provide its customers with the accurate assessment of the future stock values for the world's leading companies. In order to accomplish its goal, IEM collects financial data from many sources, analyzes them, and produces its assessments. Internally, the company has employees organized to roles according to their functions. More specifically, Data Collectors (DC) produce Collected Data (CoD), and Stock Analysts (SA) analyze CoD and produce Analytical Data (AnD). Both CoD and AnD are used by SA to produce the final assessment data. Due to the nature of its business, IEM considers the integrity of data a top priority at all times. The integrity requirements for DC and DoC are summarized as follows.

- IR1 (information-flow): Every DC and SA is assigned a trust level based on his/her records of performance and analytical accuracy. As the trust levels may change dynamically, the trust levels should be computed by using a designated function, getTrustLevel($USERID), whenever needed. Here, $USERID is a pseudo-variable representing the ID of the current user.
- IR2 (information-flow): A DC can create or modify CoD items unless his/her trust level equals 0. When a DC creates or modifies a CoD item, the trust level of the DC must be reflected on the confidence level of the CoD item. The confidence level reflects how much we can be sure about the correctness of the item.
- IR3 (data verification): CoD can be decisive factors in the stock value assessment. Thus, if the confidence level of a CoD item is less than a specific level, c, the item must be verified by a predefined verification procedure, *verifyCoD*(*this*) before it is referenced by SA.
- IR4 (information-flow): A SA may also create CoD items if it is necessary for his/her analysis, and such a CoD item's confidence level is determined by the trust level of the SA who has created it. However, in order to create CoD, a SA must have a trust level higher than a specific level, t.
- IR5 (information-flow): The confidence level of an AnD item is determined by the trust level of the SA who has created or modified the AnD item.
- IR6 (information-flow): Some SAs (whose trust levels are less than a specific level, ℓ) are in their training, and they can create AnD items, but should not modify any AnD item that has a confidence level greater than ℓ.

2) Metadata Templates and Integrity Control Policies for the Running Example: IR1 is addressed by the metadata templates *template-DC* and *template-SA* defined as follows. With these metadata template definitions, whenever a user activates either a DC or SA role, an integer type trust level is assigned to the user based on the result of the user defined function, getTrustLevel().

```
CREATE MD-TEMPLATE template-DC FOR role : DC {
    trustLevel integer : getTrustLevel($USERID);
}

CREATE MD-TEMPLATE template-SA FOR role : SA {
    trustLevel integer : getTrustLevel($USERID);
}
```

IR2 is addressed by the metadata template *template-CoD* and the ACP *ACP-IR2* defined as follows. In the ACP, *DC.trustLevel* and *CoD.confidenceLevel* represent the attribute values of the metadata items corresponding to a DC who inserts or updates a CoD item and to a CoD which is inserted or updated by the DC, respectively.

```
CREATE MD-TEMPLATE template-CoD FOR table : CoD {
    confidenceLevel integer : 0;    // a default value
    verified boolean : false;    // a default value
}

CREATE ACP ACP-IR2 FOR (CoD, DC) {
    WHEN    Insert, Update;
    IF      (DC.trustLevel ≠ 0);
    THEN    Allow: (CoD.confidenceLevel = DC.trustLevel);
    ELSE    Deny: Do Nothing;
}
```

IR3 is addressed by the DVP *DVP-IR3* and the ACP *ACP-IR3* defined in what follows. With the DVP, whenever a CoD item is about to be read, the CoD item is first verified by the specified function, and then, the result is recorded in the metadata. With the ACP, a CoD item can be read by a SA only if its confidence level is greater than or equal to c or it has been verified successfully.

```
CREATE DVP DVP-IR3 FOR CoD {
    WHEN    Read;
    IF      verifyCoD(this);
    THEN    (CoD.validated = true);
    ELSE    (CoD.validated = false);
}

CREATE ACP ACP-IR3 FOR (CoD, SA) {
    WHEN    Read;
    IF      (CoD.confidenceLevel ≥ c) OR (CoD.verified = true);
    THEN    Allow: Do Nothing;
    ELSE    Deny: Do Nothing;
}
```

IR4 is addressed by the ACP *ACP-R4* defined in what follows. With the ACP, a SA can create a CoD item only if his/her trust level is greater than t; the confidence level of the CoD item is determined by the trust level of the SA who creates it.

```
CREATE ACP ACP-R4 FOR (CoD, SA) {
    WHEN    Insert;
    IF      (SA.trustLevel > t);
```

```
    THEN    Allow: (CoD.confidenceLevel = SA.trustLevel);
    ELSE    Deny: Do Nothing;
}
```

IR5 is addressed by the metadata template *template-AnD* and the ACP *ACP-R5* defined in what follows. With the ACP, any SA can create an AnD item; the confidence level of such an item is determined by the trust level of the SA who creates it.

```
CREATE MD-TEMPLATE template-AnD FOR  table : AnD {
    confidenceLevel integer : 0; // a default value
}

CREATE ACP ACP-R5 FOR (AnD, SA) {
    WHEN    Insert;
    IF      (SA.trustLevel ≥ 0);
    THEN    Allow: (AnD.confidenceLevel = SA.trustLevel);
    ELSE    Deny: Do Nothing;
}
```

IR6 is addressed by the ACP *ACP-IR6* defined in what follows. With the ACP, a SA can modify any AnD item if his/her trust level is greater than ℓ. However, if his/her trust level is less than or equal to ℓ, then the SA can modify only AnD items with confidence levels less than or equal to ℓ.

```
CREATE ACP ACP-IR6 FOR (AnD, SA) {
    WHEN    Update;
    IF      (SA.trustLevel > ℓ) or (AnD.confidenceLevel ≤ ℓ);
    THEN    Allow: (AnD.confidenceLevel = SA.trustLevel);
    ELSE    Deny: Do Nothing;
}
```

From the examples, we can see how our integrity policy language can describe arbitrary and complex application-specific integrity requirements which cannot be captured with a single integrity model.

III. Three Alternative Integration Strategies

Implementing the integrity policy language on top of a DBMS can be seen as the integration of a new language and the DBMS. There are some alternative strategies for implementation according to which the database system should be extended. In this section we discuss three possible strategies, referred to as source code level integration, application level integration, and language level integration, respectively. As shown by Figure 1, each such strategy performs the integration at a different level. The figure also shows that the component implementing and enforcing the integrity policies, referred to as *integrity controller*, receives two types of input: (i) statements specifying the integrity policies, expressed in the language introduced in the previous section; and (ii) queries. Queries need to be checked to determine whether the query results comply with the integrity policies. For example, if a user can only access data with high integrity level, the query results need to be filtered by discarding data with low level integrity.

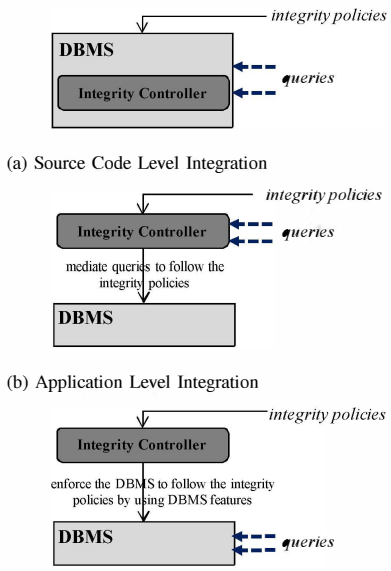

Fig. 1. Strategies for integrating integrity policies with DBMS.

The source code level integration (SLI) strategy consists of tightly coupling the integrity controller with the core DBMS code. Therefore, the integrity policy language is directly implemented by the DBMS. Such strategy is the most comprehensive and solid, but, obviously, it is the hardest to implement. Usually, the number of source code lines of commercial DBMSs is over millions and the source code is not publicly available. In addition, such strategy would require a different implementation for each DBMS, thus reducing the portability of our solution.

In the application level integration (ALI) strategy, data integrity is assured by a mediator module between the DBMS and applications. That is, the integrity controller acts as a mediator. The mediator receives as input both integrity policies and applications queries and modifies the queries or the query results according to the integrity policies. The major advantage of this strategy is that it is easy to implement. It is also DBMS independent; then the same mediator module can be used for different DBMSs. However, this strategy also has a major drawback. To assure integrity, all accesses to DBMS should be mediated by the integrity controller. If a query is submitted to the DBMS by bypassing the integrity controller, the integrity policies will not applied to the query, thus resulting in returning data that may violate the integrity policies. Because current DBMSs provide a large variety of interfaces for accessing the database (e.g., web database gateways, call level interfaces, web services, and graphical interfaces), it is very difficult to make sure that all accesses go through the mediator.

The language level integration (LLI) strategy uses the DBMS features to implement and enforce integrity policies. The main idea underlying this strategy is that the integrity

controller compiles the integrity policies into a set of objects and functions implemented by the DBMS, such as triggers, views, and auditing functions. Such objects and functions are automatically executed by the DBMS when queries and updates are issued. The LLI strategy means that the integrity language is translated into another language and thus the main task of the integrity controller is to generate DBMS-understandable statements reflecting the integrity policies described by our high level specification language. Consequently, the integrity controller acts as a compiler running on top of the DBMS. Notice that an important advantage of this strategy is that the integrity controller does not need to intercept queries and updates, since the enforcement of the integrity policies is executed by the DBMS itself. This strategy, which is the one we adopt in our work, is also the easiest to implement since it does not require changing the DBMS or the application code. The LLI strategy also provides, compared to the ALI strategy, a more secure solution in that it is more difficult for the applications to bypass the integrity enforcement. The LLI strategy can be easily ported on top of different DBMS by minor changes to the compiler for adjusting the integrity policy translation to specific features of the target DBMS.

IV. Design of the Language Level Integration (LLI) for Oracle DBMS

In this section we discuss the design of the integrity policy management system (integrity system, for short) according to the LLI strategy. We use Oracle as our target system as according to our analysis it is the system that most closely matches the requirements for the implementation of the integrity system. We first outline the architecture of the integrity system and then show how the integrity policy language is automatically translated into the DBMS language.

A. Integrity System Architecture

The integrity policy language translator (see Figure 2) is the core module of the integrity system implemented according to the LLI strategy. This module translates the integrity policy specifications described with metadata template descriptions into statements expressed by the languages supported by Oracle, such as SQL and PL/SQL. Specifically, our implementation design uses triggers for detecting events, tables for managing metadata, and VPD or FGA for simulating select triggers.

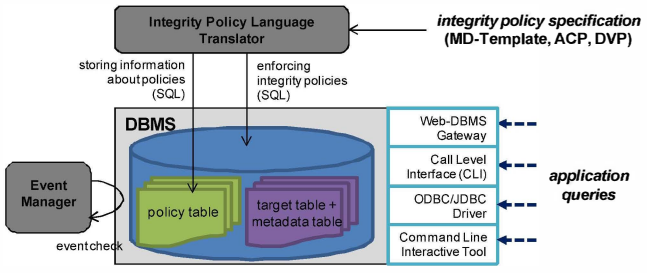

Fig. 2. System architecture of the integrity policy management system.

The translator module stores information about the policies into the policy table which is a relational table managed by the translator for managing policies registered in the system. Besides the policy tables, there are two kinds of tables in the database: 1) target tables which are defined and used by applications and 2) metadata tables which are defined by metadata template specifications and used for the integrity policy control.

The event manager module is an independent process that detects and handles the database-independent events that are specified as part of the policies expressed in the DVP specification. As we mentioned in Section II-B, the events that a DVP can specify include non-database events such as timers, counters, or specific signals. These kinds of events cannot be detected only with DBMS built-in facilities; therefore we include an event manager to periodically execute event detection.

B. Integrity Policy Language Translator

In this section, we introduce the approach for translating ACPs and DVPs onto Oracle.

1) Metadata templates: In our LLI strategy, metadata templates are represented as relational tables storing metadata items. Hereafter, we refer to a relational table created for a metadata template as *metadata table*. The integrity policy language translator automatically translates a metadata specification into SQL statements which create 1) a metadata table and 2) triggers for managing the metadata table whenever changes occur in the *target*. Table I outlines the main steps of the translation.

Step 1	Create a metadata table for the metadata template
Step 2	Insert initial tuples into the metadata table
Step 3	Create a trigger for new entries in *target*
Step 4	Create a trigger for deleted entries in *target*

TABLE I
Steps for **Create Metadata Template**

In Step 1, the schema of the metadata table is generated based on the template descriptions which consist of *attr_name*s and *attr_type*s. Along with the attributes, the metadata table also includes key attributes to map each metadata item into a specific item in *target*. When the *target_type* is 'table', a relational table is created for storing metadata of the target table. There is one-to-one mapping between data items in the metadata table and data items in the target table. It means that each item in a metadata table corresponds to a data item in a target table. For the mapping, the metadata table has the same key values of the target table. When the *target_type* is 'role', a relational table is created for storing metadata of all users in the role. There is also one-to-one mapping between users and metadata items, thus, the key of the metadata table is user_id which uniquely distinguishes users in the system.

In Step 2, if some tuples already exist in *target*, the corresponding metadata items are inserted into the metadata table. If the *target_type* is 'table', the initial metadata items for the items in the target table are inserted. The initial value of the

metadata follows the description in the *attribute_description*. If the *attribute_description* is a value, then all the initial metadata have a same value. If it is a procedure call, the value can be obtained from the results of the procedure according to the attribute values in the target table. When the *target_type* is 'role', the step is same as 'table' tuples but the initial metadata tuples are generated for the users are currently logged on the system.

In Steps 3 and 4, triggers to synchronize the *target* and its metadata table are generated. Here, 'synchronize' means that the metadata table stores tuples only for the information that is actually stored in the *target*. The triggers automatically insert or delete corresponding metadata tuples for the information which is inserted or deleted for the *target*. For the 'table' metadata template, we create an insertion trigger for generating a new metadata tuple whenever a new tuple is inserted into the target table. The initial value is decided by the *attribute_description* as in Step 2. We also create a delete trigger for removing a corresponding metadata tuple whenever a tuple is deleted from the target table. For the 'role' metadata template, we create a login trigger for generating a new metadata tuple whenever a user logs into the system. We also set a logoff trigger for removing a corresponding metadata tuple whenever a user logs off from the system.

There is an alternative way to handle deletion in the 'table' metadata template. In Step 1, after creating a table for the metadata template, we can set a cascading deletion between the target table and the metadata table. Then, the corresponding tuple in the metadata table is automatically deleted when a tuple in the target table is deleted.

2) Access Control Policies: In our LLI strategy, each ACP is implemented as a trigger. We create a trigger for an event described in the *ac_event* of a create ACP statement. In the body of the trigger, we perform access control based on the *decision* and execute the *action* according to the evaluation result of the *condition*. Triggers for insert, delete, or update events are trivial to implement since Oracle directly provides trigger facilities for these kinds of events. However, Oracle does not support select triggers and so we had to simulate these triggers by using the Virtual Private Database (VPD) and the Fine-Grained Auditing (FGA) functions of Oracle. Specifically, we use VPD for access control and FGA for the execution of the *actions*. A VPD can dynamically generate additional query conditions based on the current context of the database. Therefore, by these additional conditions, we can control that queries only access (i.e., read) the data tuples allowed by the ACPs. The FGA function allows one to execute complex statements (including SQL statements) for auditing purposes for each tuple accessed by a query. Therefore, by specifying integrity statements (instead of auditing statements), we can execute the ACP *actions*.

The automatic translation of our select triggers onto the VPD and FGA functions of Oracle is outlined in Table II.

In Step 1, the trigger is defined as a 'BEFORE' trigger since whether the operation (i.e., insert, update, or delete) can be allowed or denied is determined by the *condition* before the execution. By using PL/SQL, which is Oracle's procedural extension to the SQL database language, it is straightforward to specify the body of the trigger. The trigger evaluates the *condition* and, if the *decision* is 'DENY', raises an application error to halt the operation. For both 'DENY' and 'ALLOW' *decision*, we execute the corresponding *action* to preserve the data integrity described in the ACP statement.

Step 1	If *ac_event* is Insert, Update, or Delete, Create a trigger for the event
Step 2	If *ac_event* is Read, 1) Create a VPD for the access control described in *condition* and *decision* 2) Create a FGA for the execution of *action*

TABLE II
STEPS FOR **CREATE ACP (ACCESS CONTROL POLICY)**

3) Data Validation Policies: The major difference between ACPs and DVPs is the event which invokes the policies. An ACP is invoked when a user in a specific role accesses (i.e., read, insert, delete, or update) a specific data. However, in a DVP, the events can be independent from data accesses. As we already discussed in Section II, the events for DVP include timers, counters, or signals from outside the DBMS. We refer to these kinds of data access-independent events as user-defined events and in this section focus on how to handle user-defined events. We omit the translation rule since it is similar to the translation rule for ACPs.

User-defined events are handled by an independent process, which we refer to as the event manager. Since Oracle does not directly support user-defined events, we need to convert each user-defined event into a DBMS-known event (e.g., insertion) to then use DBMS facilities such as a trigger. The event manager in Figure 2 performs such task by periodically checking whether a user-defined event has occurred and executing an insertion into a log if the event has occurred.

V. EVALUATION

In this section, to evaluate the effectiveness of the LLI strategy, we show how very well known integrity models [2] are easily expressed in our integrity policy languages and then automatically translated onto the Oracle specific language.

A. Settings

For the evaluation, we implement two well-known data integrity models with LLI strategy: the Biba model and Low Water-Mark (LWM) model [2]. In these models, a system consists of a set S of subjects, a set O of objects, and a set I of integrity level. In our relational database setting, S is a set of users in the DBMS (or a set of roles), and O is a set of relational tuples in the database. We define a function $iL()$ to map a subject $s \in S$ or a object $o \in O$ into an integrity level $i \in I$.

The Biba model defines the following two integrity rules:

- *No read down* rule: $s \in S$ can read $o \in O$ if and only if $iL(s) \leq iL(o)$.

- *No write up* rule: $s \in S$ can write to $o \in O$ if and only if $iL(o) \leq iL(s)$.

Here, the 'no read down' rule means that a user at a given level of integrity must not read any tuple at a lower integrity level. The 'no write up rule' means that a user at a given level of integrity must not write any tuple at a higher integrity level.

The LWM model is very similar to Biba model. The LWM model uses the same 'no write up' rule of the Biba model, but it allows a subject to read data with lower integrity levels. The LWM model thus replaces the 'no read down' rule of the Biba model with the following rule:

- *Low Water-Mark* rule: If $s \in S$ reads $o \in O$, then $iL'(s) = min\{iL(s), iL(o)\}$, where $iL'(s)$ is the subject's integrity level after the read.

Here, the 'low water-mark' rule means that when a user reads a record, his/her integrity level will be changed to the minimum between the integrity level of the user and the integrity level of the record.

The application scenario is to maintain the integrity of an evidence database according to the Biba model or to the LWM model. The database consists a table *evidence* whose attributes are *evidence_id*, *title*, *content*, *category*, *owner*. Here, *evidence_id* is indexed and has a unique integer value (i.e., the primary key of the table). *title*, *owner* and *content* store the evidence information itself. *category* has a random integer value and is used for controlling the selectivity of queries (see the next paragraph).

B. Metadata Management

Because both the Biba model and the LWM model rely on the integrity levels assigned to the subjects (i.e., users) and to the objects (i.e., tuples), we need to record these levels in a metadata table. We thus create metadata templates for the evidence table and users in the database. Figure 3 shows the metadata template specifications expressed with our integrity policy language.

```
CREATE MD-TEMPLATE evi_intL FOR table : evidence {
    integrity_level number
          : initIntegrityLevelEvid(@TARGET.owner)
};

CREATE MD-TEMPLATE user_intL FOR role : all {
    integrity_level number
          : initIntegrityLevelUser(@TARGET.role)
};
```

Fig. 3. Statements expressed in our integrity policy language for creating metadata templates.

The metadata template table `evi_intL`, created for the target table `evidence`, includes only the `integrity_level` attribute, which is of integer type and records the integrity level for each tuple in the `evidence` table. The initial value of the `integrity_level` attribute is assigned by a PL/SQL procedure `initIntegrityLevelEvid()` based on the integrity level of the user who has inserted the tuple. Another metadata template, called `user_intL`, is created for the database users in order to record, for each user, the integrity level. Figure 4 shows the PL/SQL statements which are automatically generated from the metadata creation statements by our integrity policy language translator.

```
CREATE TABLE md_evi_intL (integrity_level number,
    CONSTRAINT md_evi_intL_pk PRIMARY KEY() ,
    CONSTRAINT md_evi_intL_fk FOREIGN KEY()
    REFERENCES evidence() ON DELETE CASCADE);

CREATE TABLE md_user_intL(user_id NUMBER,
    integrity_level NUMBER,
    CONSTRAINT md_user_intL_pk PRIMARY KEY(user_id))

CREATE TRIGGER md_trg_user_intL_logon
    AFTER LOGON ON DATABASE
    DECLARE v_userName VARCHAR2(128); v_userId INTEGER;
            v_roleId INTEGER; ref_attri NUMBER;
    BEGIN
        v_userName := USER;
        SELECT user_id, role INTO v_userId, v_roleId
            FROM userlist WHERE user_name = v_userName;
        SELECT role INTO ref_attri
            FROM userlist WHERE user_name = v_userName;
        INSERT INTO md_user_intL
            VALUES(v_userId,
                   initIntegrityLevelUser(ref_attri));
    END;

CREATE TRIGGER md_trg_user_intL_logoff
    BEFORE LOGOFF ON DATABASE
    DECLARE v_userName VARCHAR2(128);
            v_userId INTEGER; v_roleId INTEGER;
    BEGIN
        v_userName := USER;
        SELECT user_id, role INTO v_userId, v_roleId
        FROM userlist WHERE user_name = v_userName;
        DELETE FROM md_user_intL WHERE user_id= v_userId;
    END;
```

Fig. 4. PL/SQL statements for the creation of metadata templates in the LLI strategy.

For the `evi_intL` metadata template, a table `md_evi_intL` is created with two attributes: (`evidence_id`, `integrity_level`). Here, `evidence_id` is a foreign key referring to the `evidence_id` column of the `evidence` table; this foreign key associates each metadata tuple with a unique tuple in the `evidence` table. `integrity_level` is an attribute of type NUMBER that indicates the integrity level of the corresponding data tuple referred by `evidence_id`. The table is created with the `DELETE CASCADE` option to automatically remove the metadata tuple whenever the target tuple is removed from the `evidence` table. Another table, `md_user_intL`, is created for storing the `user_intL` metadata template. It has two attributes: `user_id` and `integrity_level`. `user_id` is the primary key of the table and uniquely identifies which user is associated with a metadata tuple. `integrity_level` indicates the integrity level of the user. Next, two triggers are created for all users. The `md_trg_user_intL_logon` trigger inserts a metadata tuple for each user in the `md_user_intL` table whenever the user logs in. The integrity level of the user is retrieved from a system table by the `initIntegrityLevelUser()` PL/SQL function. The `md_trg_user_intL_logoff` trigger deletes a tuple of a user from the metadata table `md_user_intL` whenever the user logs out.

Whenever a new data tuple is inserted in the `evidence` table, a corresponding metadata tuple is inserted into the

md_evi_intL table (see Figure 5 that shows the ACP statement for the management of this insertion).

```
CREATE ACP biba_insert FOR (evidence, all) {
    WHEN insert;
    IF   true;
    THEN allow :
        INSERT INTO @OBJECT.MD.evi_intL.integrity_level
        VALUES ( @SUBJECT.MD.user_intL.integrity_level);
    ELSE deny  : NOTHING;
};
```

Fig. 5. An ACP for managing insertions in the md_evi_intL table.

Here, the integrity_level value for the new tuple is the same as the integrity level of the user who inserts the record. This ACP statement is automatically translated by our integrity policy language translator into the PL/SQL statements shown in Figure 6. The trigger trg_acp_biba_insert inserts one tuple into md_evi_intL metadata table whenever a new tuple is inserted into the evidence table.

```
CREATE OR REPLACE TRIGGER trg_acp_biba_insert
    AFTER INSERT ON evidence FOR EACH ROW
DECLARE v_userName VARCHAR2(128); v_lvalue NUMBER;
        v_rvalue NUMBER; v_temp1 NUMBER;
        v_roleId INTEGER;
BEGIN
    v_userName := USER;
    SELECT md_user_intL.integrity_level INTO v_temp1
        FROM userlist, md_user_intL
        WHERE userlist.user_name = v_userName
        AND userlist.user_id = md_user_intL.user_id;
    INSERT INTO md_evi_intL(evidence_id,integrity_level)
        VALUES (:NEW.evidence_id, v_temp1);
END;
```

Fig. 6. SQL statements for the creation of the insert trigger.

C. No Write Up Rule

Figure 7 shows the ACP statement for specifying the 'no write up rule'.

```
CREATE ACP biba_no_write_up FOR (evidence, all) {
    WHEN update
    IF   @OBJECT.MD.evi_intL.integrity_level <=
         @SUBJECT.MD.user_intL.integrity_level;
    THEN allow : NOTHING;
    ELSE deny  : NOTHING;
};
```

Fig. 7. 'No write up' rule expressed in our integrity policy language.

The ACP is defined for the evidence table as its object and all users as its subject. When a user tries to update (i.e., write) a tuple in the table, the ACP enforces the 'no write up' rule. We can see that the 'no write up' rule of the Biba model can be easily specified by a single statement of our integrity policy language. In the statement, the rule is simply represented with a condition in the IF clause and an access control policy which permits the update only for the users who satisfy the condition. Our integrity policy language translator automatically translates the ACP statements into PL/SQL statements shown Figure 8.

As we can see from the figure, our translator generates a trigger for controlling the update on each tuple in the evidence table. The trigger compares the integrity levels of the user and the tuple. If the integrity level of the user is higher than that of the tuple, the update is admitted, otherwise it is denied.

```
CREATE OR REPLACE TRIGGER trg_acp_biba_no_write_up
    BEFORE UPDATE ON evidence FOR EACH ROW
DECLARE v_userName VARCHAR2(128); v_lvalue NUMBER;
        v_rvalue NUMBER; v_temp1 NUMBER; v_roleId INTEGER;
BEGIN
    v_userName := USER;
    SELECT integrity_level INTO v_lvalue FROM md_evi_intL
        WHERE md_evi_intL.evidence_id = :OLD.evidence_id;
    SELECT integrity_level INTO v_rvalue
        FROM md_user_intL WHERE user_id = 1;
    IF v_lvalue <= v_rvalue THEN
        doNothing;
    ELSE raise_application_error
         (error_code, 'access denied');
    END IF;
END;
```

Fig. 8. PL/SQL statements for 'no write up' rule.

D. No Read Down Rule

Figure 9 shows an ACP statement for specifying the 'no read down' rule.

```
CREATE ACP biba_no_read_down FOR (evidence, all) {
    WHEN select;
    IF @SUBJECT.MD.user_intL.integrity_level
       <= @OBJECT.MD.evi_intL.integrity_level;
    THEN allow : NOTHING;
    ELSE deny  : NOTHING;
};
```

Fig. 9. 'No read down' rule expressed in our integrity policy language.

Basically, the structure of the biba_no_read_down ACP statement is the same as that of the biba_ no_write_up ACP statement, except that the event triggering the policy is *select* instead of *update*. However, this difference makes the translation more difficult, since Oracle does not support select triggers and we have thus to simulate these triggers with the VPD and FGA functions. Here, we however only need to use the VPD since the 'no read down' rule does not require any data modification. It only includes access control for queries. Figure 10 shows the translation of the ACP statement encoding the 'no read down' rule onto PL/SQL statements.

```
ALTER TABLE evidence RENAME TO evidence_org;
CREATE VIEW evidence_view AS select * from evidence_org;
CREATE PUBLIC SYNONYM evidence FOR evidence_view;

CREATE OR REPLACE FUNCTION evidence_read
    (oowner IN VARCHAR2, ojname IN VARCHAR2)
    RETURN VARCHAR2 AS con VARCHAR2 (1024);
v_user_integrity_level NUMBER;
BEGIN
    SELECT md_user_intL.integrity_level
        INTO v_user_integrity_level
        FROM userlist, md_user_intL
        WHERE userlist.user_id = md_user_intL.user_id
        AND userlist.user_name = USER;
    con := 'evidence_id IN (select evidence_id
            from md_evi_intL
            where integrity_level >= '
                || v_user_integrity_level || ')';
    RETURN (con);
END evidence_read;

BEGIN
    DBMS_RLS.ADD_POLICY (object_name => 'evidence_org',
    policy_name => 'sp_evidence',
    policy_function => 'evidence_read',
    sec_relevant_cols => NULL);
END;
```

Fig. 10. PL/SQL statements for 'no read down' rule.

First, the translation process creates a view and a synonym for the original *evidence* table to avoid infinite recursive calls to the VPD function. The function generates an additional condition for queries to make sure the subject only accesses tuples whose integrity level is lower than the subject's integrity level.

E. Low Water-Mark Rule

Figure 11 shows the ACP statement for specifying the 'low water-mark' rule.

```
CREATE ACP lwm_integrity_revision FOR (evidence, all)
{
 WHEN select;
 IF   true;
 THEN allow:UPDATE @SUBJECT.MD.user_intL.integrity_level
           VALUES ( MIN (
           @OBJECT.MD.evi_intL.integrity_level,
           @SUBJECT.MD.user_intL.levelintegrity_level));
 ELSE deny : NOTHING;
};
```

Fig. 11. 'No read down' rule expressed in our integrity policy language.

The structure of the `lwm_integrity_revision` ACP statement is almost identical to that of the `biba_no_read_down` ACP. However the translation is more complex because it includes not only select event but also data modification to be performed during the read operation. Therefore, we simulate the select trigger with the Oracle fine-grained auditing (FGA) function. We do not use the VPD mechanism since the 'low water-mark' rule basically allows one to access any tuple in the database (i.e., there is no access control). Figure 12 shows how the ACP statement for such rule is translated onto PL/SQL statements.

As in the 'no read down' rule, the translation process first creates a view and a synonym for the original `evidence` table to avoid infinite recursive calls for the FGA procedure. The generated procedure, which is executed at run time whenever a query is issued on the `evidence` table, first executes a query which is the same as the issued query; then, it finds the minimum between the user integrity level and the tuple integrity level. Finally, it updates the integrity level of the user with such minimum value.

```
ALTER TABLE evidence RENAME TO evidence_org
CREATE VIEW evidence_view AS select * from evidence_org
CREATE PUBLIC SYNONYM evidence for evidence_view

CREATE OR REPLACE  PACKAGE audit_handler IS
    PROCEDURE HANDLE_LWM_ACCESS
    (object_schema VARCHAR2, object_name VARCHAR2,
     policy_name VARCHAR2);
END;

CREATE OR REPLACE PACKAGE BODY audit_handler IS
PROCEDURE handle_lwm_access
(object_schema VARCHAR2, object_name VARCHAR2,
 policy_name VARCHAR2)
IS PRAGMA AUTONOMOUS_TRANSACTION;
v_var1 NUMBER;
v_var2 NUMBER;
v_var3 NUMBER;
v_var4 NUMBER;
v_var5 NUMBER;
CURSOR c_results IS SELECT evidence_id FROM evidence_org
                    WHERE system.current_where;
BEGIN
    SELECT user_id INTO v_var4
        FROM userlist WHERE user_name = USER;
    OPEN c_results;
    LOOP
        FETCH c_results INTO v_var3;
        EXIT WHEN c_results\%NOTFOUND;
        SELECT integrity_level INTO v_var1
            FROM md_evi_intL WHERE evidence_id = v_var3;
        SELECT integrity_level INTO v_var2
            FROM md_user_intL WHERE user_id = v_var4;
        IF v_var1 > v_var2 THEN
            v_var5 := v_var2;
        ELSE
            v_var5 := v_var1;
        END IF;
        UPDATE md_user_intL SET integrity_level = v_var5
            WHERE user_id = v_var4;
    END LOOP;
    COMMIT;
    CLOSE c_results;
END handle_lwm_access;
END;

BEGIN
    dbms_fga.add_policy(object_name=>'evidence_view',
    policy_name=>'EVID_ACCESS_HANDLED',
    audit_column => NULL,
    audit_condition => NULL,
    handler_module =>'AUDIT_HANDLER.HANDLE_LWM_ACCESS');
END;
```

Fig. 12. Translated PL/SQL statements for 'low water-mark' rule.

VI. Conclusion

In this paper, we have proposed a policy-based approach for the definition of data integrity policies according to the requirements of collaborations. We believe that our approach is a practical solution for assuring data integrity in collaborations. As future work, we plan to develop integrity control policy languages and implementation techniques for more complex data models and access control models. We also plan to investigate how to handle conflicting policies by resolving the conflict and combining policies into a conflict-free policy before the translation.

Acknowledgements: The authors have been partially supported by Northrop Grumman as part of the NGIT Cybersecurity Research Consortium and by the NSF Grant N.0964294 "NeTS: Medium: Collaborative Research: A Comprehensive Approach for Data Quality and Provenance in Sensor Networks".

References

[1] E. Bertino and R. S. Sandhu. Database security-concepts, approaches, and challenges. *IEEE Trans. Dependable Sec. Comput.*, 2(1):2–19, 2005.
[2] K. Biba. Integrity considerations for secure computer systems. *Technical Report TR-3153, Mitre*, 1977.
[3] M. Bishop. Computer security: Art and science. *Addison-Wesley*, 2003.
[4] J.-W. Byun, Y. Sohn, and E. Bertino. Systematic control and management of data integrity. In *SACMAT*, pages 101–110, 2006.
[5] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–195, 1987.
[6] R. Ramakrishnan and J. Gehrke. Database management systems. *McGraw-Hill*, 2000.