Lifecycle Management of Relational Records for External Auditing and Regulatory Compliance

Ahmed A. Ataullah and Frank Wm. Tompa
David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada
{aataulla, fwtompa}@cs.uwaterloo.ca

Abstract— **Transactional business records are subject to a wide array of regulatory and auditing requirements. The problem of converting task specific business policies to database level constraints is challenging due to the immense complexity of corporate workflows and record lifecycles. In this paper we present a modeling framework for identifying business processes and record lifecycles within relational database systems that supports the automatic generation, implementation and verification of low level data management constraints. Our modeling language allows users to identify states of business processes within a relational database system and subsequently to enforce a broad set of conditional business rules based on the particular path that a business process has taken in the model. Our approach is unique in that it offers a single unified layer for process modeling and implementing complex workflow based constraints, temporal access control constraints, and records retention restrictions. Furthermore we propose the notion of "business process integrity" as a layer above traditional database integrity constraints, which combines conditional access control and general purpose temporal integrity constraints, to assure external auditors that each business record in the database has followed a legal path to its current state.**

Keywords: records management; relational databases; modeling integrity constraints; object lifecycle modeling.

I. INTRODUCTION

Relational database management systems, being the hub of all transactional and financial data in most large organizations, have recently received significant attention from auditors and external regulators [1]. External auditors no longer consider database systems to be black boxes that generate business reports, and it has become common practice for courts to issue subpoenas and litigation holds against records contained in corporate database systems. Furthermore external authorities can demand access to a database system and conduct a thorough forensic analysis of corporate records management practices, going as far as examining transaction logs, access control events, and even exceptions generated by a system to specific user requests. In most situations a company must comply with the request of the external authority for full disclosure, and often to provide technical assistance to auditors is the most advisable option.

In this paper we present a framework for provably demonstrating that a database system and records contained therein comply with a records management policy derived from a published business process. This problem is specifically of interest to privacy and financial auditors (both internal and external), who are responsible for verifying that corporate records management practices actually reflect the published policies that the corporation claims to follow. To provide a concrete example, let us consider a business that allows its employees to file expense claims but mandates that all such claims be approved by the filing employee's supervisor before they are paid out. In such a setting an internal auditor may wish to verify that the organization follows its own set of published rules within the context of expense claims. A traditional auditor, one dealing with non-computerized records, will most likely sample physical expense claims forms and select a few at random that look suspicious for further scrutiny. However in a situation where a corporation processes hundreds of such claims daily using its relational database system, a more savvy auditor will likely be interested in detecting violations to the above rule by querying the database to retrieve all claims where the employee is also the approver. If the result of this query is empty then the auditor can be satisfied that no expense claims that violate the above rule exist in the system.

Unfortunately this still is not adequate "proof of compliance" for the modern auditor. The focus of today's business process auditing techniques has shifted from simply identifying past violations to preventing future violations by paying specific attention to the safeguards embedded in corporate data management practices. To ensure the highest level of compliance and to satisfy technical auditors, an organization has to demonstrate that not only are there no current policy violations, but also that given the set of integrity constraints imposed on the database there can be no violations of any business rule in any conceivable database state of the database in the future.

Our proposed system aims to offer this proof of compliance directly and to demonstrate that for user specified business processes, any transaction that violates the overall "process integrity" of a workflow will be aborted and all states of the database will be compliant with the published process/records management lifecycle. In this work we will assume that the underlying database and operating systems are correctly implemented, and that their integrity cannot be compromised [2]. Our key contribution is that of proposing a database level modeling and implementation framework through which users can graphically and declaratively specify business processes within a relational database system. In our modeling framework users can identify classes of records as database level objects and subsequently define business processes and workflows by identifying the relevant changes in process specific business records. Once business level workflows have been mapped into their database level equivalents, users are able to specify conditions on these workflows graphically as transitions between the states of the relevant business records. Our system can then derive database level integrity constraints and access control restrictions that accomplish a business's overall policy objectives pertaining to the specific workflow as

well as providing a formal proof that a violation of these rules cannot occur or will be logged when it does occur. Using our framework database administrators are not only able to offer a concise snapshot of their business workflows as implemented in a database system but also to provide evidence that the constraints embedded in these workflows are being met via the integrity constraints generated by our system. To the best of our knowledge our framework is the first that attempts to offer the ability of total process management and auditing in the context of business workflows in databases systems.

The remainder of this paper is organized as follows: In Section II we describe how abstract business level workflows can be modeled in a database. We then examine implementation issues in Section III and discuss therein how our modeling language can be used to derive integrity constraints directly enforceable with a relational database setting. Section IV demonstrates how our modeling language can use transactional level metadata and auditing information to build rich conditional and temporal access control based workflows within database systems. Finally we conclude this paper by comparing our modeling framework with related work and by providing a brief summary of our contributions.

II. MODELLING BUSINESS PROCESSES AND RECORD LIFECYCLES

Business records (objects or artifacts) are essentially collections of data involved in any business process, and the notion of record lifecycle refers to the systematic and process oriented evolution of a record from creation to destruction (or permanent archival). The physical contents of business records at the database level usually have a very strong correlation with business processes with which they are associated. For example, if an expense claim is filed by an employee, it may correspond to the creation of a record in a database and the first step in the business process of handling such claims. Each stage of this process may mandate different rules and constraints on updates that can be applied against the physical contents of the record in the database.

There are several reasons why the management of business level constraints over a record's lifecycle can be challenging in relational database systems. Foremost is the fact that there is no direct support for identifying business records in database systems and enforcing integrity constraints over them. Complex business records rarely correspond to individual rows in database tables and often writing business level assertions over these records requires manual programming of integrity constraints over several tables. Similarly traditional access control and record protection mechanisms neither take into consideration the state of a business record nor the path it has taken in its lifecycle to the current state. However one of the fundamental requirements in auditing of business processes is to be able to identify the entire path a record has taken (audit trail) in its lifecycle and whether restrictions at different phases in the path were met or not. We define the overall problem of systematic management of constraints over records that can have complex paths in their lifecycle as records lifecycle management. We now present a model oriented solution to records lifecycle management in relational database systems. To the best of our knowledge our proposed framework is the first to use a process centric approach to address this problem and presents record lifecycle modeling as a means to unify integrity and access control constraints management for records in relational database systems.

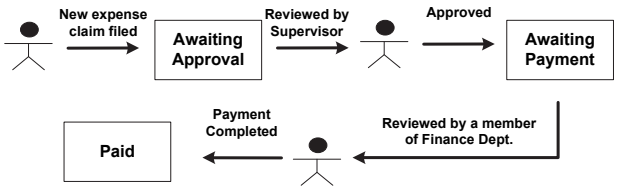


Figure 1: A high level workflow describing expense claims management process

A. *Identifying Business Records*

Before policies over record lifecycles can be enforced, we need to address the challenge of identifying records contained in a database system that will be involved in a process based workflow. Business records such as invoices, expense claims and sales reports typically have an intuitive meaning for managers and everyday users. Unfortunately in relational databases these everyday records may correspond to results of complex parametric queries whose underlying data may be spread across many tables in a database system. In our work we adopt the object based view of business records in database systems, where a record is defined as a uniquely identifiable tuple in a relational *view* [3][4] (and not necessarily to a tuple in a base table). We require that the actual definition of the business object involved in a workflow must come from the user (administrator or auditor) in the form of a relational view. This view will define the scope and attributes of a particular class of business records and will be considered as the *object definition* or *record definition* for policy purposes. Each tuple in this view will represent an individual *object* or *record* of the specified type, and we will use these terms interchangeably to refer to a unique row in the specified view.

In order to model the business process related to expense claims, the user must provide a view definition *E*, that identifies all expense claims in the database system. Each tuple in *E* will represent an individual expense claim object and must be uniquely identifiable via a combination of attributes in the view. We also note that these object definitions can be of vastly different degree of query complexity. For example the view *E* defining expense claim objects can be as simple as a selection of a table or it can be view defined using more expressive query operators such as joins and aggregations of multiple tables.

B. *Record States*

Once users have defined object types, the next step in modeling their lifecycle is to identify policy-relevant states that those objects can take. We mentioned earlier that record lifecycles have a direct correlation with business workflows, and each data event (creation, modification and deletion of a record) signifies progression at some level in a business process. Consequently defining the various states that a relational record can take is simply a matter of giving meaning within a database system to the states in a business process. To motivate this with our running example, let us consider the

Transition	Restriction
 Exit Restriction	If a record was in state A at time *t*, and does not satisfy the condition to be in state A at time *t+1*, then it must satisfy the condition to be in state B at time *t+1*.
 Entry Restriction	If a record is in state B at time *t*, and does not satisfy the condition to be in state B at time *t-1*, then it must satisfy the condition to be in state A at time *t-1*.
 Never Eventually Transition	If a record was in state A, it should never subsequently satisfy the condition to be in state B.

Table 1: Basic set of transitions that establish restrictions between states

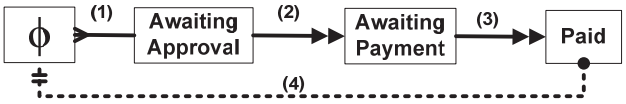


Figure 2: Constraint diagram depicting the restrictions on the legal paths an object can take during its lifecycle

simplistic workflow for processing expense claims presented in Figure 1.

Observe that the business record being acted upon in this workflow is an expense claim and that each state (rectangle) corresponds to a possible state of a particular expense claim. Since our framework considers each object to be an individual tuple in a view, it defines the notion of a state slightly differently. A *state* S is a boolean condition specified on the attributes of an object type. A record is said to be in state S if its attributes satisfy that condition. To elaborate further with our example, let us assume that a user has defined a view for expense claims with the following attributes:

E = (CLAIM_ID, EMP_ID, DATE, DETAILS, APPROVED_FLAG, AMT_CLAIMED, AMT_APPROVED, AMT_PAID, PAID_DATE)

The state *AwaitingApproval* might be defined as the condition "APPROVED_FLAG = FALSE". Each record *r* in the view *E* that satisfies this condition will be said to be in the *AwaitingApproval* state. Similarly we can define the other relevant states in our workflow using boolean conditions such as *AwaitingPayment* and *Paid* with the associated conditions "APPROVED_FLAG = TRUE AND AMT_PAID = 0" and "AMT_PAID > 0" respectively. For notational convenience we will consider the state name itself to be a boolean function on the tuples of a record. For example an expense claim record *r* will be said to be in the *AwaitingApproval* state if and only if *AwaitingApproval*(*r*) is true.

Note that users are also free to identify intermediate states that may not be present in higher level workflows being modeled in the database system. Since workflows are high level descriptions, they contain minimal information about the data contained to manage and implement constraints on a business record. Furthermore it may be necessary to drill down into a simplified workflow of a business process and identify policy relevant sub-states within states mentioned in the workflow. Since a state is simply a condition on the attributes of a view, there could potentially be an infinite number of conditions that can be listed for record lifecycle modeling. However we rely here on the user (administrator or auditor) to provide only a list of conditions on objects that are relevant to the monitoring and enforcement of their own policy objectives. In general a record (tuple in a user defined view) can be in multiple states at any given point in time because it can potentially satisfy more than one state conditional at the same time. Therefore a record could potentially traverse multiple paths during its lifecycle.

C. Constraint Diagrams and Record Lifecycles

Once users have specified the states involved in a workflow as relevant conditions over tuples in record defining views, they can then identify conditional (path) constraints over the record lifecycle depicted by the original workflow. We now present our record lifecycle modeling language with which users can build constraint diagrams to enforce business rules over records throughout their lifecycle. Note that a business level workflow does not explicitly specify when a record is created at the database level. Similarly a workflow may provide guidance into how a business process can move forward, but it does not explicitly identify transitions (or changes in attributes of a record) that move the database level record lifecycle towards its conclusion. Therefore to model record lifecycles in database systems we need to provide a richer set of semantics in the modeling language, so that it can support decision making and policy enforcement at the level of row and attribute deletions, insertions and updates. Because of the close association of our record lifecycle modeling with business workflows, our models will mirror workflows in the states that they encapsulate, but they will typically contain a richer set of transitions and will have state specific constraints embedded in them. We will therefore denote our record lifecycle models as constraint diagrams.

Constraint diagrams are a simplified model of restricting the ways in which a record can move between states in its lifecycle. Each transition in the model will represent a constraint on the path that a record can take from one state to the other. We propose three basic transitions that are listed in Table 1, but we allow others to be defined as needed [5].

Furthermore we define a global state, ϕ, to signify a non-existent record. Transitions to the state ϕ imply that a record is no longer part of the view defined by the user (it has been deleted) and transitions originating from the state ϕ imply insertion of a tuple in the view (signifying the creation of a new object). With all the basic building blocks in place we can present a model of our original workflow as a constraint diagram (Figure 2).

Observe that although the constraint diagram we present looks similar to the workflow in Figure 1 there is significantly greater information embedded in the constraint diagram. For example transition (1) specifies the constraint that any new tuple inserted in the view must satisfy the condition to be in the state *AwaitingApproval.* In other words for all expense claim records *r* created, *AwaitingApproval*(*r*) must be true, a function which requires that the condition on the attribute APPROVED_FLAG of *r* must initially be false. From a

lifecycle perspective this transition restricts how objects can be created, but in terms of the underlying business objects it also enforces the rule that any new expense claim records that are introduced must not have already been paid or approved. The reciprocal of transition (1) is transition (4). It attempts to place a restriction on deletions initiated from the *Paid* state by imposing a 'never eventually transition' modeling the constraint that once an expense claim has been paid out, it should never be deleted. This models part of a typical records retention constraint for records of interest to financial auditors. Transitions (2) and (3) are entry restrictions on the states that they reach and ensure that an object that reaches the states *AwaitingPayment* or *Paid* must have come from the states *AwaitingApproval* or *AwaitingPayment*, respectively. The reason for using an entry restriction instead of exit restriction is because not all expense claims may get approved. However a typical auditing rule would require that all expense claims that do get approved must have been awaiting approval first and must have gone through the approval process.

III. From Modeling to Implementation

A. State Oriented Audit Trails

The concept of keeping track of the paths taken by an object during its lifecycle is essentially that of maintaining an audit trail of the record and tracing it from the beginning (creation) through the current system time. Most modern relational database systems have built-in support for auditing (via triggers or materialized views) and many organizations maintain audit trails of sensitive business records. Our constraint management system can utilize existing audit trails to identify paths of various business records without any additional storage overhead. A typical example of an audit table in a database is as follows:

ObjectValues = (OBJ_ID, timestamp, OBJ_Attribute1, OBJ_Attribute2, … , OBJ_AttributeN, *user*, *txn_type*, *user_group*, *purpose*, …)

Each row in the audit table represents the complete details of an object, identified by OBJ_ID at a particular timestamp. Each row may also contain other information including the *user* who last modified the object, the *user group* to which that user belonged and auxiliary information such as the *purpose* of the transaction initiated by that user. The amount and granularity of the auxiliary information is application dependent and may include any auditing relevant metadata about the transaction that modified the object on that timestamp.

We view the audit history of an object from a state based perspective and introduce the concept of a *state transition history*. By examining the ObjectValues we have the full details of an object and all its attributes available, and therefore we also have the ability to determine which state(s) an object belongs to at a particular point in time. We can do this by simply checking each state conditional against the attribute values of each history entry to determine if the object belonged to a particular state at a given time. Consequently if we have N user defined policy relevant states (boolean functions) for a particular object, we can view the state transition history as a sequence of object states:

ObjectStateValues: (OBJ_ID, timestamp, S_1,S_2,…S_N)

where each S_i is a boolean value representing whether an object is in state S_i at the given timestamp. Using such an abstraction we can simplify reasoning over the object's past history, being concerned not with the actual data values that an object held at a particular point in time but whether those values corresponded to an object being in a state of interest. We will denote the binary string represented by the concatenation of S_1, S_2,…S_N as a *state configuration* which will give us an instant snapshot of which states an object belongs to (by there being a 1 in their respective positions for those states).

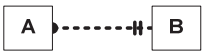
Transition	Restriction
 Exit Restriction	$\bullet A(r) \wedge \neg A(r) \Rightarrow B(r)$
 Entry Restriction	$B(r) \wedge \neg \bullet B(r) \Rightarrow \bullet A(r)$
 Never Eventually Transition	$\blacklozenge A(r) \Rightarrow \neg B(r)$

Table 2: Converting diagrammatic constraints into first order temporal logical integrity constraints

B. Logical Equivalence of Transitions and Temporal Integrity Constraints

With a simplified model of the object's state transitioning history available we can now lay the ground work for the implementation of restrictions on objects embedded in constraint diagrams. Observe that each constraint (transition) in the diagram is a temporal restriction involving two states and their respective conditionals. Our goal in this section is to demonstrate how each of these transitions can be converted into run-time rules and database level assertions to prevent record modifications that are deemed illegal by the record lifecycle model.

Each of the restrictions in a constraint diagram can, by design, be directly translated into temporal formulas (implications) involving two states conditionals. We use the following classical definitions of temporal operators [6] to reach a compact logical representation of transitional constraints:

1. Previously (●): If A is a first order temporal formula then ●A is true at time $t > 0$ if and only if A is true at time *t-1*
2. Sometime in the past (♦): If A is a first order temporal formula then ♦A is true at time *t* if and only if there exists a time $k < t$ when A was true

Table 2 provides a summary of each state based restriction and how they directly translate into first order temporal logic based integrity assertions. Note once again that we rely on the encapsulation of a state name as a function to make the formulas more succinct.

Once we establish this one-to-one mapping between transitions in our constraint diagram and temporal assertions, we have a succinct and formal representations of the constraints (state transitions) specified in the relational object lifecycle model. Looking back at Figure 2 we can now see that transition (2) represents the constraint "*AwaitingPayment*(r) ∧ ¬●*AwaitingPayment*(r) ⇒ ●*AwaitingApproval*(r)" and transition (4) is logically equivalent to ♦*Paid*(r) ⇒ ¬ ϕ(r). We can also define the notion of workflow integrity in terms of user initiated transactions. A transaction is said to be *business process integrity preserving*, if after committing, it leaves the database in a state where all integrity constraints derived from user created constraint diagrams are satisfied.

C. Multistate Path Constraints

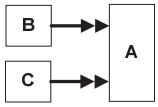


Figure 3: A constraint that if an object reaches state A, it must have previously been in states B and C simultaneously

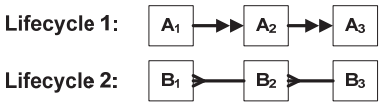


Figure 4: Multiple models of how an object can behave during different phases in its lifecycle can be independently created, and temporal assertions can be enforced independent of other modeling constraints.

One of the advantages, as well as a source of analytical complexity, in our modeling language arises from the fact that objects can enter and exit multiple states asynchronously. Figure 3 gives an example of a constraint diagram in which an object is required to trace multistate paths. Observe that while all records in state A must at some point have been in both states B and C at the same time, the two temporal assertions derived from the model, A(r) ∧ ¬●A(r) ⇒●B(r) and A(r) ∧ ¬●A(r) ⇒ ●C(r), are independent of each other and both must independently hold true. Although there may be situations where B and C overlap and can be collapsed into one state, we believe that this independent enforcement of each temporal assertion has significant benefits.

One of those benefits is that different stakeholders in an organization can describe their own relevant portions of the object lifecycles. Furthermore they do so without being limited by other stakeholders' constraints in any way. Observe that based on the constraints presented in Figure 4, given a single update operation, an object can fail (or begin) to satisfy conditions to be in different states across both lifecycles. For example, an object that is initially in states A_2 and B_1, can after an update, be legally present in states A_3 and B_2, or states A_3 and B_1, or states A_2 and B_2, or not change state configuration at all because of the update. The ability of objects to traverse multiple paths in different lifecycles at the same time allows us to give a much broader meaning to the notion of an object's lifecycle. We can use our system to model various intermediate phases in an object's lifecycle without (necessarily) attempting to combine them in a single diagram. Figure 5 presents an

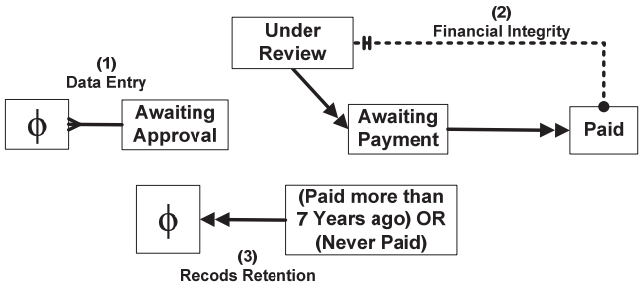


Figure 5: Individual constraint diagrams do not necessarily need to model all aspect of an object's lifecycle.

extended version of our running example in this paper. The diagram presented illustrates how three different departments of a business, may have specified three different constraint diagrams on the same object. In fact it may very well be the case that the states labeled "Awaiting Approval" and "Under Review" are logically equivalent or that the sub-process associated with transitioning expense claims from "Awaiting approval" to "Under Review" is stated elsewhere by another department of the business.

Using the above distributed method of developing sub-process oriented constraint diagrams, many stakeholders can contribute towards the overall constraint management of a business process. We emphasize that, in general, not only can each component (of a lifecycle model such as the one in Figure 5) be viewed separately as a valid constraint diagram but at the most basic level each transition between two states can be treated as a diagram representing an assertion that must be met at every update to an object. If an object lies in several states then all assertions associated with those states will have to be checked at every update.

D. Enforcement

The elegance of our modeling language lies in the fact that, by design, lifecycle modeling of business objects allows users to specify complex temporal assertions about data objects of interest without having the need to specify these assertions using first order logic. However another key benefit of our framework is that not only can it be employed for auditing business processes and object lifecycles in relational systems, it can also be used for enforcement of the specific rules being audited.

Temporal integrity constraints have been exhaustively examined within the database research community. The works of Chomicki and Toman [7][8] discuss mechanisms for optimizing and enforcing temporal assertions within relational databases. For brevity we do not present the full details of their methodology. Rather we point out that integrity constraints derived from our model, are directly equivalent to first order temporal logic formulas expressed over the state configuration history of an object in PAST-TL (temporal logic of the past), as discussed in Chomicki's seminal work on integrity constraints. Since each database object will have a finite amount of history, the problem of checking compliance with individual temporal assertions is immediately understood to be decidable. Active enforcement is simply a matter of implementing triggers on views to ensure that for each update to the database, the

transaction making the changes should not be allowed to commit unless the newly derived state transition history for all objects affected, complies with all the temporal integrity constraints embedded in the database system [5]. These triggers can be directly derived from the techniques presented by Chomicki and Toman [8].

The continuous online checking of integrity constraints can impose a significant computational overhead, because in order to reason about individual constraints we must calculate the set of states an object *will belong* to given a change to its attributes. This computation of the new (to-be) state configuration of objects has to be done for every transaction that modifies any object of interest. If, for all impacted objects, the new state configuration passes all temporal assertions derived from our model it is allowed to commit. We note that this cost of continuous monitoring has to be incurred regardless of whether a modeling framework is utilized to generate low level integrity constraints or not. However a modeling framework such as ours can lead to optimizations being discovered within the modeled lifecycles while at the same time reduce the total cost of the implementation when business processes are themselves evolving.

E. Auditing

One of the ways in which we can minimize the above discussed overhead for online-enforcement of temporal assertions, and at the same time provide a means for loosely maintaining business process integrity is through automated and periodic auditing. However before we proceed we need to examine the auditing implications of our modeling framework in greater detail to build a case for delayed checking of business process integrity.

Our modeling of temporal integrity constraints revolving around a model of a business process allows us to examine auditing and enforcement together as a single problem. To elaborate further, let us consider a database system that does not enforce any temporal business process integrity constraints and for which an auditor wishes to verify corporate business practices at the data object level. The auditor can utilize our model and run the history of each object through a set of temporal integrity constraints derived from constraint diagrams. The auditor would effectively be using our modeling language to create a process-centric lifecycle of records that s/he believes the organization claims to follow. Then the auditor would "run" the audit logs of objects of interest as virtual simulations over constraints derived in our models to see whether and where in the lifecycle of a particular object an action took place that violated the overall workflow integrity of the database.

System administrators can, in principal, perform this self-audit of historical records periodically in an attempt to identify already committed violations so that they can be rectified before an actual audit takes place. The primary benefit of this approach is that these self-audits can be performed when the system is under low load conditions, or even be conducted on a separate database system with a copy of the transactional data. The disadvantage is that the violations that this method will detect would have already been committed by the time they are discovered. Thus we consider this method to be loosely-compliant with business process integrity. If post discovery corrective actions are not taken by system administrators, then the liability associated with not actually following the particular business process will still be present. Furthermore there will be a verifiable proof of non-compliance (at least one object not following a properly mandated lifecycle) physically present to incriminate an organization.

For brevity we omit a discussion of how database administrators may choose to correct the violations committed in the past as there may be numerous ways an organization may be able to do so. However we recognize that often correcting past violations may not be feasible and therefore propose a hybrid approach to enforcement and self-auditing. In general businesses may have the flexibility to isolate mission-critical process level constraints and those that can be loosely checked. In practical situations we see administrators specifying a priority level for each of the restrictions (transitions) in the model to signify which ones need strict enforcement and which ones should be detected by periodic self-audits. This approach offers a much needed balance between run-time efficiency and process integrity, which the database administrators can fine-tune themselves based on the business requirements.

IV. RICHER CONSTRAINT DIAGRAMS

One of the key benefits of our constraint modeling framework is that it is simple enough for business users to flesh out object lifecycles at the database level without sacrificing the expressivity provided by first order temporal logic. So far we have examined object states that pertain only to the object's intrinsic data values. However many business processes may also rely on external meta-data such as information about the users who modified the particular object and the conditions associated with those external modifications. We now examine how the identification of rich states of records involved in a business process can accomplish broad ranging policy goals for a business.

A. Temporal access control constraints

Recall from Section III.A that a typical auditing table in a database contains significant amount of metadata associated with the transaction being logged. Transactional metadata such as the user who initiated a transaction and the purpose of a transaction is typically of great value to auditors, and many business processes rely on knowledge about users, purposes, and types of transaction to move these processes forward in different directions. As an example we can consider a scenario where a business process may require that all expense claims of over $50 in value must be approved by a member of the Finance department. In database terms, users need to be able to specify conditions associated with transactions by which an object was updated.

To support such decision making in business process integrity monitoring, system administrators and auditors only need to specify state conditions that utilize this transaction related metadata. For example we had previously defined the *Paid* state for all expense claim objects to be such that AMT_PAID > 0. A variant state such as *PaidHighAmount*

could be described as AMT_PAID > 50 AND *txn_type="Payment" AND user_group="Finance"*. The interpretation given to an expense claim object being in such a state is simply that it satisfies the condition to be in the *Paid* state and also that it was brought to this state via a transaction of type *payment* and the user who initiated that transaction belonged to *Finance* department.

The direct benefit of this approach to augmenting state conditions with information available in the auditing metadata is that we can now encapsulate very rich business paths within the object lifecycle. Temporal access control restrictions are generally known to be difficult to model. However using our lifecycle model in which the state information about an object or a business process encapsulates access control events, we can bypass the need to have an access control model that is separate from the business process model. Moreover complex conditional constraints such as, "Expense claims can only be paid out by employees in the Finance department, if they have previously been approved by an employee of the Administration department," that depend on specific authorizations (transactions) and actions taken by specific users in the past, can very easily be modeled and enforced by re-examining the audit trail and state transition history.

B. Records Retention

The overall objectives of any corporate records retention policy are twofold: to protect records when mandated by legal requirements or for auditing purposes, and to destroy (or anonymize) records when they have lived their useful life. Our framework can be directly used to model data retention policies and can even go as far as providing a means for verifying (auditing) whether the policy being modeled has been implemented correctly. Note that our monitoring framework itself cannot delete records, anonymize data or make any data oriented changes to a database system to accomplish the objectives of a data retention policy. However, just as we made the case for auditing and loose compliance, our framework can certainly be used to detect when records exist in a particular state such that they need to be purged or protected because of legal requirements.

A simple method of protecting records until they have met their minimum retention obligations can be accomplished in our system by disallowing transitions to state ϕ. Transition (4) in Figure 2 is one such example that protects expense claim objects from deletion once they are in the *paid* state. Although preventing a transition to state ϕ prevents deletion of objects, it does not offer fine grained control over which attributes of a record can (or cannot) be modified by users. For example an administrator may want to allow modifications to the *comments* associated with an object in the *paid* state but at the same time disallow changes to other fields. There are several ways we can extend our model to offer fine grained retention restrictions. One such method would be to create a special state that disallows modifications unless the given update complies with a user specified condition on the attributes. Conditions such as disallowing updates to certain attributes can be accommodated as long as the user is able to annotate these in the constraint diagram. For brevity we do not formally present diagrammatic semantics that can be employed for this purpose. However we do emphasize that once users have built basic object lifecycle models using our presented framework, protecting records against conditional updates is simply a matter of additional annotations (conditions) on the diagram associated with specific states. These annotations in a state based scenario would simply provide additional information (conditions) to the underlying assertion monitoring system in order to allow or reject a given database transaction.

The complementary problem associated with records retention, of destroying or anonymizing records when they have outlived their useful life, has to be handled somewhat differently. As mentioned earlier our proposed system can only be used to monitor the states of objects and cannot physically perform deletion for the user or automatically modify objects and thus move them into different states. Furthermore we need to acknowledge that there may be multiple ways that can be used to remove objects from a view or move them out of a state that requires destruction.

However, we believe that from a system administrator's perspective, monitoring can make enforcement easier by immediately identifying records that belong to a state that requires them to be purged according to a retention policy. For example let us say that an organization wishes to destroy expense claims that are in a particular state. As an object reaches this state in its lifecycle, our system can notify the administrator, or even automatically execute a pre-written procedure to purge the object. Furthermore the system can also determine whether execution of the procedure caused the object to be removed from the database system or, in the case of anonymization, moved to a different pre-specified privacy preserving state.

We conclude this section by observing that modeling retention constraints is simply a matter of recognizing which states are relevant to an object's records retention policy and how subsequent actions should change those objects. Although, verifying that the actions executed by the system actually accomplish the intent of the data retention policy is an unsolvable problem [4], we are still able to offer a modeling language that makes it easy to implement and verify the execution of the retention actions on objects, and subsequently reason about the lifecycles being followed by individual objects.

V. RELATED WORK

Our approach to constraint modeling for database systems is related to many business process modeling (BPM), object lifecycle modeling, database constraint management, and systems verification techniques examined in the literature. Here we provide a brief description of how our work relates to the large amount of published work in the above areas.

Our modeling language distinguishes itself by focusing on modeling changes in data objects (rows in relational views) as business processes progress toward their conclusions. Modeling changes in object values bears close resemblance to the problem of assertion checking in the formal verification of reactive software systems. Various formulations of problems in the field of model checking, and formal methods for software verification, have attempted to build models for checking paths

that should never be traversed in the execution of program code [9]. Our approach also resembles and, to a large degree, complements the process modeling efforts at higher levels within an organization, whether they be artifact oriented [10][11][12], object lifecycle oriented [13][14], or based on classical methods such as analyses of state based petri-nets and workflows [15][16]. Since constraints in a database system can generally be considered as event-condition-action (ECA) based rules, there is substantial amount of work done related to the analysis and implementation of ECA rules in database management systems that relates to ours, and we direct the reader to [17] for a survey of the area.

The key contribution of our work goes beyond modeling and is that of establishing a relationship between the modeling language and database level integrity constraints. Prior attempts to provide a constraint specification and management framework for relational systems, such as Object Constraint Language [18], and ER^2 modeling [19] have significant shortcomings that our approach addresses. The existing models of constraint modeling lack the ability to have arbitrarily expressive views for identifying complex data objects and are unable to specify conditional temporal constraints. Furthermore prior approaches have largely adopted a static (snapshot based) view of data objects that diverges from the business process based object lifecycle modeling methodology that has been widely accepted by the business community.

VI. Conclusion

In this paper we presented an expressive and extensible database constraint management framework. Our technique of constraint management relies on generating first order temporal assertions created from constraint diagrams that represent business process models within database systems. Our primary contribution in this paper is that of proposing a language for modeling business processes within relational database systems that allows for easily generating complex temporal constraints for active enforcement.

Using our approach, system administrators and auditors can diagrammatically specify relational object lifecycles to implement integrity constraints that ensure compliance with the overall goals of the business policy/process being modeled. Using the available transaction meta-data users can very easily express temporal access control constraints that not only address the roles of individual users executing specific transactions but also the purpose of each transaction being taken in light of a business process. It is our belief that the proposed modeling framework significantly reduces the policy design, implementation, and auditing overhead costs incurred in the presence of a large number of complex business process-based policies originating from various functional areas of a business.

References

[1] T. McCollum, "Data analysis with SQL," Internal Auditor, Aug. 2006.

[2] S. Mitra, M. Winslett, R. T. Snodgrass, S. Yaduvanshi, and S. Ambokar, "An architecture for regulatory compliant database management," Proc IEEE International Conference on Data Engineering (ICDE '09), 2009, pp 162-173.

[3] A. Adya, J. A. Blakeley, S. Melnik, and S. Muralidhar, "Anatomy of the ADO.NET entity framework," Proc. ACM Special Interest Group on Management Of Data (SIGMOD '07), 2007, pp. 877-888.

[4] A. Ataullah, A. Aboulnaga, and F. W. Tompa, "Records retention in relational database systems," Proc. ACM Conference on Information and Knowledge Management, (CIKM '08), 2008, pp. 873-882.

[5] A. Ataullah, and F. W. Tompa, "Business Policy Modeling and Enforcement in Databases," unpublished, www.cs.uwaterloo.ca/~aataulla/unpublished/BPM_in_Databases.pdf.

[6] D. M. Gabbay, I. Hodkinson, and M. Reynolds, Temporal Logic: Mathematical Foundations and Computational Aspects. Oxford University Press, 1994.

[7] J. Chomicki, "Efficient checking of temporal integrity constraints using bounded history encoding," Transactions on Database Systems. vol. 20, no. 2, Jun. 1995, pp. 149-186.

[8] J. Chomicki and D. Toman, "Implementing temporal integrity constraints using an cctive DBMS," IEEE Transactions on Knowledge and Data Engineering, (August 1995), pp 566-582

[9] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking, The MIT Press, 1st edition, Jan. 2000.

[10] K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su, "Towards formal analysis of artifact-centric business process model," *Proc.* 5th International Conference on Business Process Management (BPM'07), 2007, pp 288-304.

[11] R. Hull, "Artifact-centric business process models: Brief survey of research results and challenges," Proc. OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE, 2008, pp. 1152-1163.

[12] A. Deutsch, R. Hull, F. Patrizi, and C. Vianu, "Automatic verification of data-centric business processes," Proc. 12th International Conference on Database Theory (ICDT '09), 2009, pp.252-267.

[13] K. Ryndina, J.M. Küster, H. Gall, "Consistency of business process models and object life cycles," MoDELS 2006. LNCS, vol. 4364, pp. 80–90.

[14] G. Kappel and M. Schrefl, "Object behavior diagrams," Proc. International Conference on Data Engineering, (ICDE '91), 1991, pp 530-539.

[15] E. Oren and A Haller, "Formal frameworks for workflow modelling," Digital Enterprise Research Institute, National University of Ireland, Technical Report 2005-04-07, April 2005.

[16] K. Knorr, "Dynamic access control through petri net workflows," Proc. 16th Annual Computer Security Applications Conference, (ACSAC '00), 2000, pp 159-168.

[17] N. W. Paton and O Diaz, "Active database systems," ACM Computing Surveys, vol. 31, no. 1, March 1999.

[18] B. Demuth, H. Huβmann, and D. Loecher, "OCL as a specification language for business rules in database applications," Proc. 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, (UML '01), 2001, pp. 104-117.

[19] A. K. Tanaka, "On conceptual design of active databases," Doctoral Thesis. UMI Order Number: UMI Order No. GAX93-15916., Georgia Institute of Technology, United States of America, 1993.