Business Policy Modeling and Enforcement in Databases

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

ABSTRACT

Database systems are the central information repositories for businesses and are subject to a wide array of policies, rules and requirements. The envelope of business level constraints implemented within database systems has expanded from classical access control to include auditing, usage control, privacy management, and records retention. The lack of a systematic mechanism of integrating and reasoning about such a diverse set of policies manifested as database level constraints makes corporate policy management a chaotic process.

In this paper we propose a general purpose policy modeling and constraint management framework that can integrate numerous aspects of business level requirements within database systems. Our proposed solution relies on a finite state modeling language for business level policies, in which users can declaratively express rules related to the normal workflow of a business process as well as specifying any undesirable states of business objects contained in a database system. The proposed system is then able to translate these policies into low level temporal integrity constraints that prevent policy violations and ensure that business objects and artifacts follow their mandated lifecycles. A formal layer for reasoning allows policy makers to discover unenforceable and conflicting policies, providing the basis to guaranteeing compliance for a wide array of rules that may need to be enforced on complex business objects stored in relational database systems.

1. INTRODUCTION

Database systems are subject to a wide variety of continuously evolving constraints that originate from business level policies. However in the absence of a mechanism that bridges the gap between the database administrator's view of the world (constraints, queries, and triggers) and the corporate policy officer's view of the world (business rule, objects, and workflow conditions) the management of a large number of rules imposed over a database system becomes extremely challenging. The end result is typically a complex web of integrity constraints and hand written triggers derived from business requirements imposed by different operational areas of the business [5][9].

The absence of a centralized policy and constraint management system within traditional database systems leads to several problems, the foremost being *lack of transparency.* With a large number of interdependent business rules and an equally large number of constraints in a database system, it becomes difficult to manage and audit whether the overall goals of a business policy are being met within the set of implemented constraints. Whether these constraints are functional dependencies or manually written triggers, it becomes extremely challenging to identify particular policy level objectives being accomplished by each one. Conversely, identifying the set of constraints that are relevant to a particular policy becomes equally difficult, and these hurdles can impose significant maintenance costs. Furthermore ad-hoc implementation of business policies can significantly reduce the overall *manageability* of the system. One of the goals of this work is to allow privileged users (enforcement officers who are not necessarily database administrators) to specify business level rules using a simple and easy to understand language and then automatically enforce the translated integrity constraints within the corporate relational database system. By decentralizing policy creation there exists the opportunity to reduce the management overhead imposed on a database administrator by removing the need to implement database level constraints manually.

Most importantly, without a formal layer for reasoning with constraints, the set of policies for which *compliance guarantees* can be offered is very limited. Modern auditors can often demand a significant level of assurance (a meta-proof) that the claimed business policies are being executed and enforced at the database level. Satisfying the technically savvy auditor in such cases means more than simply demonstrating that there are no policy violations in the current database state. The focus of the modern process auditing techniques has shifted to verifying rules that are in place to ensure business policies are being met, and we believe that in the future it will be very likely that comprehensive process audits will include analyses of database level constraints. Classical integrity constraints have the shortcoming of being able to express only a very limited class of business rules, and more complex temporal business policies often force administrators to implement an intricate set of Event-Condition-Action (ECA) based rules. The increasing awareness of data legislation imposed on DBMSs and the need for reliable and provable enforcement methods naturally brings up questions regarding confluence, termination, and enforceability guarantees of these rules. Unfortunately analyzing and reasoning about hundreds of hand-crafted triggers on various parts of the database is simply infeasible.

The core contribution of this work is to show that a policy modeling framework can addresses the problems of *transparency*, *manageability*, and *compliance* when business rules are embedded within a database system as generalized constraints. As we shall discuss in Section 2, there are several modeling techniques for business policies that are currently being used. However none of these languages are able to reason at the implementation level and offer a mechanism that combines the work done in active rule processing with policy design and constraint management. Our approach bridges this gap between the policy/business rule level and the database query/constraint level to present a framework that provides significant compliance guarantees over a broad class of business rules mapped as data level constraints. We show how to convert high level business workflows and object lifecycle restrictions into database level constraints and to use classical results to reason about their execution to detect run-time policy conflicts preemptively, allowing the policy administrator(s) the ability to minimize run-time exceptions.

The structure of the paper is as follows: In Section 2 we provide a comprehensive review of past literature on the topic of modeling policies in database systems and discuss why prior proposed models are unable to capture modern day business policy requirements. In Section 3 we present our own framework, and in Section 4 we demonstrate its extensibility and expressivity. Finally in Sections 5 we demonstrate how policies can be validated and checked for internal consistency, and in Section 6 discuss the performance of the constraints derived from policy models of varying complexity.

2. RELATED WORK

Our work lies at the intersection of business policy modeling and database integrity constraints. Modeling of business rules within database systems has been examined in many contexts over the past four decades. We shall discuss in detail the published literature that directly addresses our problem and provide only a general overview of the various topics that touch on policy and constraint management in databases. The techniques proposed for rule modeling can be classified into purely logic based models, purely graphical models, and hybrid models. The overall objective of policy modeling is to have a representation of business policy that is both easy to visualize and easy to implement.

The most well known technique for expressing simple business rules within relational systems is via physical design restrictions. UML [23] and ER diagrams [6] provide users with the ability to embed domain and integrity constraints to cater for many operational and storage level assertions. However, with the solidification of the SQL-3 standard for triggers and the prevalence of complex Event-Condition-Action (ECA) style rules being used in the policy modeling community, triggers and assertions have seen widespread adoption as a mechanism to enforce business rules [5][15]. Cochrane et al. [9] provide a detailed description of how triggers are implemented, and their work summarizes the trigger related features offered by most commercial DBMSs. It is interesting to note that, even if not explicitly employed by administrators, triggers are still widely used constructs, often transparently placed by a database system to monitor integrity violations and for the incremental maintenance of materialized views. Specific contributions for the many active rule systems such as Ariel, Starburst, SAMOS, HiPAC and ODE are too detailed to enumerate in this paper. A comprehensive survey of trigger based rule systems, and their specific termination, confluence and determinism properties is available [17]. In short, the theoretical foundations for the analysis of rule systems and triggers within database systems are well grounded, and their use in policy and assertion monitoring is ever more popular.

Graphical modeling of rules has seen significant interest from the software engineering and business policy modeling communities. UML (Unified Modeling Language) is commonly used for modeling objects as well as relational databases. It has recently been extended to include the Object Constraint Language [23] by which simple constraints on objects can be specified. However the actual enforcement of these constraints is left to the implementation. Demuth et al. [11] have proposed an extension to OCL for automatic translation of object level constraints in the modeling language to database level triggers, provided "object-to-table" mappings are available. Badaway and Richta [4] presented a similar technique and Zimbrão et al. proposed an OCL constraint to SQL assertion translation mechanism in their work [22]. Tanaka et al. took a slightly different approach to graphical modeling, introducing an extension to entity-relationship modeling (ER^2) that allows users to use diagrams to model events related to entities and conditions under which these events can take place [21][14].

Constraints on objects in the above models are inseparable from the notion of object invariants (universally true statements for all business objects of a specific type) and can rarely capture complex policies. Specifically, policies that involve temporal constructs, or repetitive application of rules, can neither be modeled by nor translated into implementation specific actions using these techniques. Because of their static nature, these modeling languages do not store the state of an object nor keep track of the history of past occurrences of a rule being applied. For a example, consider an object called "employee" with an integer valued property called "salary" and a constraint that states that an employee's salary can only be increased by a maximum of 10% in any increment. While this policy itself can be implemented via a simple constraint and can be modeled well, a more complex policy stating that an "employee's salary cannot increase by more than 25% over any three consecutive increments" cannot be modeled in a language that does not represent intermediate states of individual objects (employees) between rule applications.

The intent of currently proposed modeling techniques is to offer a simplified world view of business rules for the database programmer and the policy designer. However if a graphical modeling language is not capable of expressing constraints of a reasonable complexity or does not support a direct implementation path in a database system, then its usability is seriously diminished. When examining proposed graphical models in the context of relational database systems, it becomes apparent that unless the rules being modeled are extremely simple and straightforward (in terms of "object-to-table" mappings [4][11]), high level modeling does not make the task of writing and implementing triggers and database level constraints any simpler. Instead the simplification is usually for the benefit of business policy administrators, who are able to get a bird's eye view of the system design. The benefits of such models are largely superficial for the database programmers, who still have to determine how the constraints can be implemented and manually attempt to simplify and combine them for efficiency and manageability. Furthermore the notion of "objects" in ECA rule modeling does not acknowledge that relational objects might not always be easily identifiable in a database and that a data model used to describe business rules might not be accurately represented by an optimized (normalized) database schema. Consequently it is unclear whether these techniques have seen any adoption by database administrators and how well these model work in practice.

The apparent difference between the world view of business policy makers and database administrators has received some mention in the literature. <Add aritifact centric>

More recently we have observed that complex business records, such as invoices and sales reports, are often viewed as objects by policy makers, but the data contained therein may reflect the

execution of a complex set of queries involving temporal parameters [2]. Consequently any reasonably powerful policy modeling system that is able to express constraints over business records stored in a database should be aware of not only the underlying database schema but also the actual data definitions and the queries used to generate these records.

There is clearly a tradeoff between low level specifications and higher level models; the closer a rule model is to lower level database constructs such as queries, integrity constraints, and triggers, the more straightforward it is to deploy; and a model that is more abstract will generally be ambiguous but easier for policy makers to create, exchange, and comprehend. In this paper we propose a modeling language that strikes a balance in this spectrum, with its goal being seamless automatic generation of database level ECA constraints as well as ease of use for policy administrators.

We must emphasize that our contribution is not to offer yet another rule modeling language. The tens of business process modeling techniques such as state transition diagrams, flowcharts, data flow modeling, and work flow modeling are sufficiently adequate in their own context. However because of the lack of a modeling language that uses database constructs, we have proposed a model and graphical semantics for translating business level workflow to database (object) level workflows [3]. In this paper our aim is to demonstrate how workflow level restrictions (paths in business processes and data objects lifecycles) can be translated, implemented, and optimized over a relational database system using first order temporal integrity constraints. Our notion of policy is critically dependent on the state of an object and is fundamentally different from a traditional object level constraint as considered in prior work. While we offer graphical semantics for modeling, we emphasize that these can easily be adopted and integrated into any generic modeling language with support for database systems or object level modeling.

3. POLICY MODELING CONSTRUCTS

In order to circumvent the problem of establishing "object-to-table mappings" [11] our model requires policy level objects to be declaratively defined on the underlying database schema. More specifically, policy relevant business objects can only be defined as a logical mapping, that is, a business object in our model is defined as a tuple in a relational *view* over a database with a fixed schema. This is the fundamental difference between UML/OCL modeling and our proposed framework in that we adopt a bottom up approach in modeling constraints using database level constructs. We rely on users to identify records of interest, thereby eliminating the need for mapping policy level objects to database level objects. Although relational views could also be used to accomplish other goals, for instance to identify complex business artifacts [10], we consider the term "*view*" and the term "*object-definition*" to be the same and interchangeable.

3.1 Relational Objects and Object States

A relational view essentially defines a particular type of object (its attributes and data types) and is similar to a UML object definition. Tuples contained in the view will be called "objects" of that particular type. A *state* is identified by a label assigned to a condition on the attributes of an object definition. An object belongs to a particular labeled state if its attributes satisfy the condition associated with the state. In general the state of an object is dependent on the particular assignment of its attributes. For example let us say x is an object of type V or equivalently $x = (a_1,a_2\ldots a_n) \in V$, where V is a view over our database schema; then for a labeled state S with its appropriate conditional function, if $S(a_1,a_2,\ldots a_n)$ is true then x is said to be in state S. To simplify our notation we will consider $S(x)$ as the boolean result (true or false) of applying the state condition to the attributes of the object x.

For a particular object definition V, $S_V = \{S_{V1},S_{V2}\ldots,S_{Vn}\}$is the set of all user defined states for tuples in V. In essence each of the S_{V1} through is S_{Vn} can be considered a function that returns a true/false value for any object of type V. At any given point in time an object may belong to multiple labeled states if it satisfies more than one of these state conditionals. To quickly determine which states an object x of type V belongs to, we will denote the *state configuration* of x, $SC(x)$ to be a canonically ordered binary string of length $|S_V|$. A zero at position i in the state configuration of an object x implies that $S_{Vi}(x)$ is false and that the object does not belong to state S_{Vi}. Similarly a one at position i in the state configuration of an object x implies that $S_{Vi}(x)$ is true and that the object belongs to state S_{Vi}. Note that a state configuration of all zeros does not imply that the object does not exist but rather that it is in none of the user specified states S_V. Such a state configuration implies that the object is not currently constrained by any policy relevant to the users of the system.

A *state change* for an object refers to a change in one of the state conditions. Since a single update may change the results of several conditionals, an object can leave and enter many states between two consecutive timestamps. Thus a state change is generally evident by the state configuration of an object at two consecutive timestamps being unequal. A *policy* in our model is a (multi) state restriction or path constraint on the sequence of state changes that we will show to be equivalent to a temporal integrity constraint modeled in past temporal logic [18]. Rule execution (state transition monitoring) for policy enforcement is done on a tuple-by-tuple (object) basis, and database transactions that lead to undesirable sequence of state changes for tuples/objects are rejected.

3.1.1 Example

Let us say a business user defines the following view to capture a typical business object (an invoice) in a database system:

Invoice = (INV_ID, CREATE_DATE, PAID, AMOUNT, PAID_DATE, LATE_FEE)

The user further defines two states, S1 to signify an invoice being paid as the function S_1: PAID = *true* and S2 to represent invoices that are high valued as S_2: AMOUNT > *1000*. The flexibility of our model allows many users to simultaneously define states that they require for policy based decision making. For example a different user could independently define a state with a more complex conditional function such as S_3: *DateDifference(now* - CREATE_DATE*)* < *7* to capture the condition of a invoice being created recently (i.e within the past 7 days).

This abstraction allows us to reason about the policy relevant conditions that an object meets at a particular point in time without being concerned with the actual attributes values. Assuming that the above S_1, S_2 and S_3 are the only states defined system-wide for policy modeling on invoices, a tuple in the view with the state configuration of (0,1,1) denotes a recently created,

unpaid invoice with an amount greater than $1000. Moving from one state configuration, say from (0,1,1) at timestamp t_1 to (1,1,1) at timestamp t_2, is an event that signifies the payment of a low valued invoice within seven days of its creation. It may be helpful for readers familiar with formal verification and model checking [12] to visualize changes in the state configuration as a temporally ordered changes in the state space $\{(0,0,0),\ldots,(1,1,1)\}$.

In general, events can be logical (simple passage of time, for example, without any change to the underlying data) or user initiated data physical modifications to the data. Policies are considered to be multipath restrictions specified in linear temporal logic (LTL) over the state configuration space that an object takes. The reader should observe that there is duality between the state configuration changes and how a business object evolves over its lifespan and, in a given workflow, the business level interpretation of a particular state configuration change is implicitly provided by the state conditionals.

3.2 Data and Constraint Model

To support the modeling of temporal integrity constraints we consider a model of data where the history of every policy relevant object is maintained by the system and accessible for decision making. This is a standard assumption in situations where the decision to reject/allow updates (integrity checking) requires the examination of past attribute values of an object. To better explain our modeling language we make the simplifying assumption that instead of a typical audit trail $x_t=(t,a_1,a_2\ldots a_n)$, where each a_i represents an attribute of the object at time *t*, we will instead view the audit trail for objects as $x_t=(t, SC(x_t))$. More specifically, instead of being concerned with the attribute values that an object held at a particular time stamp, we will only be concerned with the list of states that the object belongs to at a particular point in time. This simplified view of the audit trail of the object will be called the *state configuration history*.

Although we present a diagrammatic method of specifying constraints over business objects there is a strict one-to-one correspondence between state transitions in our models and logical implications (assertions) specified in first order temporal logic of the past over the state configuration history of objects. To explain our model we rely on two classical temporal operators [18]:

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*

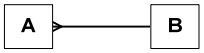
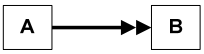
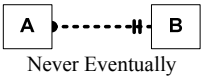
State Representation	Logical Restriction
 Exit Restriction	$(\bullet A(x) \wedge \neg A(x)) \Rightarrow B(x)$
 Entry Restriction	$(B(x) \wedge \neg \bullet B(x)) \Rightarrow \bullet A(x)$
 Never Eventually Transition	$\blacklozenge A(x) \Rightarrow \neg B(x)$
 Disallow Exit	$\bullet A(x) \Rightarrow A(x)$

Table 1: Basic set of state transitions and temporal assertions

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

However before we fully establish the duality between state transitions and integrity constraints specified in first order temporal logic (FOTLICs) let us build an example that uses our framework and demonstrates its usability.

3.3 Example

A business penalizes customers for late payments on their invoices, and an invoice is considered late if it is not paid within 30 days of it being created. A company policy dictates that payments are only accepted in full, and if a payment for an invoice is received late, a late fee is recorded on the invoice and carried forward. The accounting department of the company requires the deletion of paid invoices that were created more than seven years ago, but the customer relations department wants to ensure that details of invoices that were paid late by customers are never deleted.

3.4 Modeling Constructs

We will use the definition of the invoice object as discussed earlier in Section 3.1. A state *NewAndUnpaid* can be defined as "*DateDifference*(*now* – CREATE_DATE) < *30* AND PAID = *false*". Observe that the state is simply a testing condition and an invoice object belongs to this state if satisfies the given condition. Also note that we require policy makers to define for us both the business object as a view and all the policy relevant states as conditions using the attributes of the view. By having both object definitions and necessary conditions to identify objects present in particular states we can subsequently tie state transitions together as restrictions between how an object is (or is not) allowed to progress in a given workflow.

As a simple example, consider a state "*paid"* with its associated condition "PAID = *true*". A business level constraint that requires that once an invoice is paid, it cannot be "unpaid" is essentially equivalent to the temporal assertion $\bullet paid(x) \Rightarrow paid(x)$, and it ensures that if an object was in the paid state in the prior timestamp, it must also be paid in the current timestamp. In a typical ECA based system, for every update to the object this assertion must hold true. If this assertion is violated the transaction updating the object is rejected and rolled back.

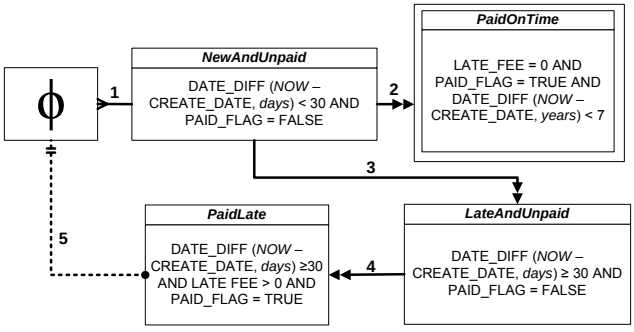


Figure 1: A constraint diagram where each transition specifies a logical temporal assertion between two states of an object.

To cover all aspects of an object's lifecycle a special state ϕ is introduced to signify moving to/from "nothingness". Transitions from ϕ to any state S refer to insertions in the view such that the newly inserted objects satisfy the state conditional of S.

Symmetrically, transitions to ϕ from any state can be considered deletions (removal from the view). Although ϕ can be considered as a "null state configuration," it is different from a state configuration containing all zeros: an object in state ϕ does not exist, whereas an object with an all-zero state configuration does exist, but it does not meet the conditions to be in any user defined state. To start, we propose four state oriented constraints in Table 1 with their logical temporal assertions shown alongside.

Using these transitions and appropriately defined states, a policy maker can easily create a diagram that depicts the policy objectives presented in Section 3.3. Figure 1 attempts to do so and capture the business requirements for the previously described invoice management scenario. We have included three states in addition to the previously defined *NewAndUnpaid* and established five basic restrictions among them. The model presented in Figure 1 is simply a means of restricting how an object can behave during different stages in its lifecycle. The logical interpretation of constraint 1 is that all newly created objects must satisfy the condition to be in the *NewandUnpaid* state. Observe that this interpretation only has meaning in the context of the given view. The reader is reminded that a new row becoming part of a view may not always correspond to a new object being physically created in the system. That correspondence is strictly dependent on the view (object) definition. Constraint 5 requires that once an object reaches the *PaidLate* state, it must never be removed from the view. Any transaction that attempts to remove (from the view) an object that has reached this state in the past is rejected. Constraints 2,3 and 4 are much simpler in that they restrict the arrival of an object in a particular state to specific states in the immediately preceding timestamp. Such constraints lay out a strict path that an object must follow in its workflow.

Every constraint diagram can be broken down to a set of temporal implications between the conditions specified by the states. For example, constraint 2, which mandates that all invoices that begin to satisfy the *PaidOnTime* condition must in the immediately preceeding timestamp have been satisfying the *NewandUnpaid* condition. We can easily use Table 1 as a lookup to provide a logical representation of the constraint as: $PaidOnTime\ (x) \wedge \neg\bullet PaidOnTime\ (x) \Rightarrow \bullet NewandUnpaid(x)$. Each of the implications is of the form $precondition(x) \Rightarrow postcondition(x)$ and there is clearly no need for a single administrator to define every object and/or state conditions in a single diagram. For example to

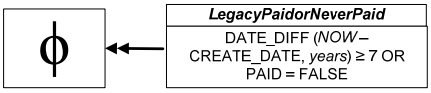


Figure 2: A constraint diagram where and object is only allowed to be removed from the view if it was in the specified state

accomplish the final retention objectives of our scenario a different policy maker could separately draw out a single restriction as shown in Figure 2 to ensure that any record is to leave the view if it was either never paid or paid more than 7 years ago. Finally we note that because of the bijection between first order temporal integrity constraints and logical restriction in the graphical model, users can also see their constraint diagrams as a simple list of their temporal assertions of the form $precondition(x) \Rightarrow postcondition(x)$ specified over the state configuration of the object.

The choice of viewing all state oriented restrictions and detailed state conditions on an object definition in a single diagram, as separate graph components, or as a list of logical assertions in first order temporal logic, is left to the user, and the ability to change from one to the other is a strength of the model. From an operational perspective, we believe, that not only will policy makers benefit from a company-wide unified definition of business objects in a relational database, but also that such definitions already exist within high level corporate workflows. Business users are generally aware of the necessary conditions for objects to be in particular states and can independently map complex workflows that are pertinent to their business functions. Whether they choose to focus on a particular path in the corporate level object workflow to model or the association between two arbitrary states depends largely on what policy objectives they are trying to accomplish.

3.5 Multipath policy restrictions

In our examples an invoice can be in multiple states at the same point in time. Specifically an invoice in the *LegacyPaidorNeverPaid* state (Figure 2) can also satisfy the conditions associated with the state *LateandUnpaid* (Figure 1). As a consequence objects can traverse multiple conceptual lifecycles at the same time, allowing multiple workflow paths to be simultaneously traversed if necessitated by policy level requirements. This complicates the enforcement model conceptually in that when an object in states START = {SS1, SS2,…SSn} moves atomically (in a single time stamp) to states END = {ES1, ES2,…ESn}, then all transitional paths traversed between these states (START × END, the cross product of these states) are subject to policy rules and restrictions implied by all models in which these states were defined. However the most significant benefit of this approach is the added flexibility for any policy maker to define his or her policy independently from other users of the system.

3.6 Enforceability

A classical problem associated with temporal integrity is that of enforceability. In general, forcing a temporal function to remain constant over time is not possible, and the use of temporal logic brings the risk of introducing policies that may not be enforceable. For example, let us say that a user has defined a state condition that returns *true* for all invoices created in the last seven days. In general any newly created invoice will only stay in this state for seven days. Therefore whether we want it or not, by simple passage of time, an invoice created today will no longer be in that state seven days later. Consequently attempts to prevent transiting out of this state are not enforceable. In our work we focus on implementing only enforceable policies against user initiated transactions and provide brief discussion of detecting non-enforceable policies in Section 5.

4. Modeling Complex Workflows

Given that we have established a mechanism by which simple state oriented restrictions can be tied together by FOTLICs, we introduce a few extensions to this model that demonstrate its expressivity. We shall shortly see that these extensions are by no means definitive or limiting, and users are virtually free to model any arbitrary temporal formula that they wish to use for constraining an object's lifecycle.

4.1 Path Constraints and Sub-formulas

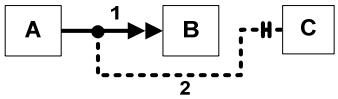


Figure 3: Constraint 1 being used as a "state" or a condition that needs to have happened in the passed

A common requirement in enforcing workflow policies is that of specifying conditional paths that require a sequence of events to happen. For example consider the constraint, "an object should never reach state C, if it has sometime in the past reached state B from state A."

Graphically we propose that this be modeled as generalizing a transition into a state. Observe that transitions, as introduced in Table 1, are rules specified on how objects must change. However we can also consider these rules as requirements that must have happened in the past to make other rules. Consider constraint (2) in Figure 3 as an example. It uses constraint (1), $B(x) \wedge \neg\bullet B(x) \Rightarrow \bullet A(x)$, as a sub-formula to specify a conditional constraint that essentially requires that if the transition from A to B took place for an object in the past, then that object should then never reach state C. More formally (2) specifies the following temporal assertion:

$$\blacklozenge(B(x) \wedge \neg\bullet B(x)) \Rightarrow \neg C(x)$$

This elegance of FOTLICs allows the use of a conditional assertion as a sub-condition to create larger more complex assertions and to repeat the process as many times as necessary to create conditional paths in policies or workflows.

4.2 Exploiting Auditing Meta-data

In most databases where audit trails of business records are being maintained, there is also significant auditing related transactional meta-data that is kept alongside. This metadata can include information about the user who initiated the update, the purpose and the context of the transaction that changed the database contents. Consequently object-level audit trails can usually be viewed as more robust event logs that not only contain the attribute values of the object being modified but several additional attributes pertaining to the transaction itself:

x_t= (timstamp,a_1,a_2…a_n, *user*, *user_group*, *purpose, application_context, transaction_type, txn_starttime* …)

An example of the benefits of transactional meta-data being available for policy-level decision making is that we can model constraints that require conditional access control based on specific workflow paths being traversed by an object. Consider an example of a policy, in which an invoice can only be modified to be in the paid state by a user who belongs to the *finance* user group, and only so, if it has previously been marked as approved for payment by an employee in the *admin* user group.

Traditional modeling of such a business rule would require significant transaction and/or application level logic to be implemented. However using our model, all a policy maker would have to do is to modify the state conditions such that they include restrictions on the transaction related meta-data to accomplish their objectives. For example he or she could easily redefine the state "paid" and add the additional constraint that the user group of the user moving an object to this state must be *finance* as an additional condition to being in that state.

Modeling access temporal control at the business work-flow level has significant advantages as it allows complex access control conditions (and sub-conditions as exemplified in Section 4.1) to be very easily understood by policy makers. Furthermore our framework provides a direct implementation path for such workflow based access control requirements by modeling them to traditional integrity constraints specified on the object meta-data trail. The implementation still enforces what can be considered a classical temporal integrity constraint, by rejecting a transaction that does not comply with the assertions derived from our model, but the end result would accomplish the objectives of conditional or workflow based access control modeling.

5. Conflict Detection and Policy Optimization

Figure 4: An indirectly specified containment restriction

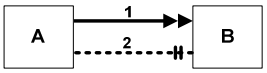


Figure 5: Temporal inconsistency (dead end)

Using FOTLICs as the logical representation for enforcement of business level policies allows us to reduce policy verification and validation to well known problems related to static query analysis. For example, let us say that (for one reason or the other) the condition specified by a policy maker that defines state S is not satisfiable. Consequently there cannot exist an object *x* that satisfies $S(x)$, thereby making the state redundant and all related constraints removable from the system of derived logical assertions.

Since the graphical model and its logical representation have a strict one-to-one correspondence, any reasoning that can be applied to first order temporal formulas, can also be reflected in the graphical model and vice versa. Consider the graphical constraint model presented in Figure 4. Constraint (1) specifies the assertion that $B(x) \wedge \neg\bullet B(x) \Rightarrow \bullet A(x)$, while constraint (2) specifies the assertion $\bullet C(x) \wedge \neg C(x) \Rightarrow B(x)$. If we analyze these implications we come to the conclusion that an object that leaves state C must arrive in state B and any object that arrives in state B must have previously been in state A, and therefore, objects in state B (that arrived through state C) must have previously been in both state A and state C at the same time. If the analysis of the conditions of A and C shows otherwise (for example that $A \cap C = \phi$) then we have an error in the form of a logical inconsistency, that makes state B unreachable, and it should be reported to the policy maker.

Another example is presented in Figure 5, where constraint (1) mandates that all objects in A when exiting the state must reach state B, $\bullet A(x) \Rightarrow B(x)$, and (2) specifies that after reaching state A, an object should never reach state B, $B(x) \Rightarrow \neg\blacklozenge A(x)$. By chaining the implications we get, $\bullet A(x) \Rightarrow \neg\blacklozenge A(x)$, which itself is an inconsistent temporal formula and represents a dead end in the workflow for the object if it ever reaches state A. In general such inconsistencies are typically the result of badly designed or conflicting policy requirements.

Two important observations about logical reasoning that concern business level policy modeling need to be made here. First we note that it is unlikely that any single policy maker will make

egregious modeling errors such as the ones described above. This is because each individual constraint diagram will most likely be a mapping of a business process that does not suffer from these problems. However as constraint diagrams from various policy makers are merged together to create a final implementation model (set of logical assertions), the need to resolve such conflicts by automated reasoning is ever present. Furthermore there are significant avenues of optimization through eliminating and combining constraints derived by different policy makers in a business to ensure the resulting set of assertions being continually checked in a database system is minimal.

Second, we note that our ability to reason over these logical constraints is directly restricted by the view (object) definitions and their query complexity, as well as the domains over which these queries are specified. For arbitrary queries and state conditions, the problems of determining containment, closure, and intersection are well known to be undecideable. However we believe that a significant number of business object definitions will involve the use of simple conjunctive queries. Logical analysis of assertions in these situations will not only be a tractable problem but will also lead to significant optimization and reduction of the number of constraints that need to be checked per transaction. A complete discussion of the computational cost and tools available to optimize a given set of assertions in first order logic is beyond the scope of this paper. However it is important to note that this optimization is a one-time cost and needs to be incurred only when a new constraint is introduced in the system. We conclude by reminding the reader that our objective in this section was to demonstrate how logical analysis of first order formulas can not only lead to conflict detection and implementation level optimizations but also graphical simplifications in the model (collapsing, removal, and separation of states) that may not be obvious when a large number of constraint from various sources are integrated.

6. Implementation and Performance

6.1 Generating ECA rules

The lack of direct support for temporal integrity constraints within commercial database systems invariably requires that such constraints be implemented as active rules or triggers. The most widely cited reference implementation of first order temporal integrity constraints implemented as SQL triggers, that reject (roll-back) transactions if a first order temporal integrity constraint is not met, was presented by Chomicki and Toman [8]. The work introduced the use of result memoization for complex formulas, through storing the results of sub-formulas, to check past temporal integrity constraints in O(1) time and avoid examining the entire history of an object.

The class of past temporal restrictions that our graphical framework model generates falls within the class of temporal integrity constraints that Chomicki and Toman showed to be preservable in O(1) time at an extra space cost of O(f) where f is the length of the formula involved (See Appendix A for details). Recall that in our model we viewed the object's history as a time-stamped sequence of binary strings (state configurations), and each entry at position *i* of a state configuration, denoted by a zero or one whether an object belonged to state S_i at a given time or not. If we consider this historical view of the state transitions made by an object to be readily available for querying, and treat each temporal assertions specified by the model as a query, then checking the validity of the assertion is simply a matter of comparing two bit strings representing the previous and the current (to be committed) state configurations of an object, every time it is updated. In fact any implementation of temporal integrity constraints will very closely resemble the check-constraint functionality provided by modern DBMS systems where the actual constraint is specified over the state transition history of the object type being modified.

6.2 Performance Analysis

Due to space restrictions we offer only a summary of our test results in this section and invite the reader to see Appendix B for a comprehensive examination of the various costs associated with temporal integrity constraints derived using our model in synthetically created business scenarios of varying complexity.

The overhead of implementing integrity constraints derived from our model can be split into space and computational costs. In our tests we were able to demonstrate that for typical business objects, specifically invoices as specified by the TPC-H benchmark, the worst case cost of storing and maintaining a state transition history has virtually no impact in a high-update transactional database. Even in the most extreme case business workflows with 1024 distinct states, the additional overhead of storing the state configuration alongside the audit trail was minimal. This is simply because the size of a 1024 bit state configuration history pales in comparison to the size of typical business objects that contain large text/comment fields. As for the computational costs, a system that already incurs the cost of additional disk writes while maintaining an audit trail will incur a comparatively insignificant cost of performing 1024 in-memory arithmetic/string operations (such as *status = "paid"*) that determine the current/new state configuration to be stored alongside the audit trail.

Finally we note that there may be situations where a database system cannot tolerate even the most insignificant of performance penalties (storage or computations). Our framework can still be used to audit the database periodically by running the audit logs (i.e. creating the state transition history) over a copy of the database and verifying it over the set of constraints generated by a model written by an auditor. This process would essentially simulate the states explored by each business object in the system and identify violations that have taken place since the last full audit of the system. Although this method cannot prevent invalid transitions (because they were already executed), the core benefit in this would be that of making the process of business process auditing much simpler and it offers the possibility of simulating the results of a change in business policy before constraints are actively enforced.

7. Conclusion

In this paper we presented a policy framework for database systems that relies on modeling temporal integrity constraints as relationships between states contained in a business level workflow. Our methodology resembles that of model checking and formal verification by examining how business objects change over time through state transitions. The most significant benefit of our framework is that it allows policy makers to map out their corporate workflows for business objects stored in relational systems and seamlessly implement constraints to enforce their process centric requirements.

Our objective was to demonstrate through the use of examples, how assertions specified in first order temporal logic can be applied to a graphical representation of a state oriented business workflow. Policy enforcers may extend this model by introducing generalized temporal operators reflecting their preferred interpretation of state transition restrictions while still maintaining a direct path to efficient implementation of integrity constraint monitoring triggers. Most importantly by using our framework businesses can seamlessly convert their existing object lifecycles and workflows diagrams, that may involve temporal and conditional access control requirements, into database level constraint with very little effort. It is because of this extensibility, expressivity, and ease of use provided by our framework that we believe that it can used effectively to mitigate the problems associated with management of business policy within database systems.

8. REFERENCES

[1] Amghar, Y., Meziane, M., and Flory, A. 2002. Using business rules within a design process of active databases. In *Data Warehousing and Web Engineering*, IRM Press, 161-184.

[2] Ataullah, A. A., Aboulnaga, A., and Tompa, F. W. 2008. Records retention in relational database systems. In *CIKM 2008*. ACM, 873-882.

[3] Ataullah, A. A., Tompa, F. W. 2011. Lifecycle management of Relational Record for External Auditing and Regulatory Compliance. In *IEEE Symposium on Policies for Distributed Systems and Networks* (*to appear*), (June 5-7, 2011). www.cs.uwaterloo.ca/~aataulla/POLICY2011.pdf

[4] Badaway, M. and Richta, K. Deriving Triggers from UML/OCL Specification. Information Systems Development: Advances in Methodologies, Components and Management, 2003. 305-315.

[5] Ceri, S., Cochrane, R., and Widom, J. 2000. Practical Applications of Triggers and Constraints: Success and Lingering Issues. In *VLDB 2000*, 254-262.

[6] Chen, P. P. 1976. The entity-relationship model—toward a unified view of data. *ACM TODS* 1, 1 (Mar. 1976), 9-36.

[7] Chomicki, J. 1995. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM TODS.* 20, 2 (Jun. 1995), 149-186.

[8] Chomicki, J. and Toman, D. 1995. Implementing Temporal Integrity Constraints Using an Active DBMS. *IEEE Trans. on Knowl. and Data Eng.* 7, 4 (Aug. 1995), 566-582.

[9] Cochrane, R., Pirahesh, H., and Mattos, N. M. 1996. Integrating Triggers and Declarative Constraints in SQL Database Systems. In *VLDB 1996*, 567-578.

[10] Cohn, D., and Hull, R. 2009. Business Artifacts: A Data-centric Approach to Modeling Business Operations and Processes. *IEEE Data Eng. Bull.* volume 32, 3-9.

[11] Demuth, B., Hussmann, H. and Loecher, S. OCL as a Specification Language for Business Rules in Database Applications. In «UML» 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools. ISBN: 978-3-540-42667-7. 114-117, 2001.

[12] Edmund M. Clarke, Jr., Orna Grumberg and Doron A. Peled, *Model Checking*, , MIT Press, 1999.

[13] Gehani, N. H., Jagadish, H. V., and Shmueli, O. 1992. Composite Event Specification in Active Databases: Model & Implementation. In *VLDB 1992*, 327-338.

[14] S. B. Navathe, A. K. Tanaka, and S. Chakravarthy. Active Database Modelling and Design Tools: Issues, Approach and Architecture. *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases*, 15(1-4): 6-9, 1992.

[15] Simon, E. and Dittrich, A. K. 1995. Promises and Realities of Active Database Systems. In *VLDB 1995,* 642-653.

[16] J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kauffmann, San Francisco, 1995.

[17] Paton, N. W. and Díaz, O. 1999. Active database systems. *ACM Computing Surveys* 31, 1 (Mar. 1999), 63-103.

[18] Prior, A. 1967. Past, Present and Future, *Oxford University Press.*

[19] Ram, S. and Khatri, V. 2005. A comprehensive framework for modeling set-based business rules during conceptual database design. *Information Systems* 30, 2 (Apr. 2005), 89-118.

[20] Spencer, B. 1998. Business Rules vs. Database Rules - A Position Statement. In *Workshop on on Object-Oriented Technology* (July 20 - 24, 1998). Lecture Notes In Computer Science, vol. 1543. Springer-Verlag, London, 200-201.

[21] Tanaka, A. K. 1992 *On Conceptual Design of Active Databases.* Doctoral Thesis. UMI Order Number: UMI Order No. GAX93-15916., Georgia Institute of Technology.

[22] Zimbrão, G., Miranda1, R., de Souza, J., Estolano, M.H, Neto, F. P., Enforcement of Business Rules in Relational Databases Using Constraints. In *Advances in Databases and Information Systems*, 2009, LNCS5968.

[23] UML/OCL v2.0 specifications available at: http://www.omg.org/technology/documents/formal/ocl.htm.

Appendix A

Equivalence of queries, constraints and assertions

Constraints specified in first order logic over a set of attributes can be written in several different ways. The methodology adopted in [7] is that of specifying a negated existential formula of the form: $\neg (\exists x)$ (*formula*(x)), or simply that there should not exist an object x in the relation that satisfies *formula(x)*. Checking this constraint is simply a matter of executing a query to find tuples that satisfy *formula*(x). If a transaction takes a database into state where there exists a tuple x such that *formula*(x) is true, then the transaction should be rejected.

In our model we consider constraints on business objects of the form: *precondition_formula*(x) $\Rightarrow$ *postcondition_formula*(x), or simply that all objects which satisfy the precondition must also

satisfy the given post condition. Formally we can write the constraint as:

$$(\forall x)\ precondition\_formula(x) \Rightarrow postcondition\_formula(x)$$

and subsequently rewrite the implication as a disjunction:

$$(\forall x)\ postcondition\_formula(x) \ \vee \neg\ precondition\_formula(x)$$

Finally we can negate the disjunction and write the original assertion as a negated existential formula:

$$\neg\,(\exists x)\,\neg\,(postcondition\_formula(x) \ \vee \neg\ precondition\_formula(x))$$

$$\neg\,(\exists x)\,\neg\,postcondition\_formula(x) \ \wedge \ \neg\ precondition\_formula(x)$$

Consequently any assertion derived from our model over the state configuration history has a direct path to execution as a SQL query for integrity constraint checking.

Alternative interpretations to reaching a state

Observe that many policy designers might interpret "reaching the state E" as E(x) being true and E(x) being previously false that is, E(x) $\wedge$ $\neg\bullet$E(x). Furthermore transitions "from" one of the start states $\{S_1,S_2,\ldots S_n\}$ could also have alternative interpretations such as "at least one of state conditions is previously true and currently false": $(\bullet S_1(x) \wedge \neg S_1(x)) \vee \ \ldots \ \vee (\bullet S_n(x) \wedge \neg S_n(x))$. These interpretations given to "exiting" and "entering" can simplify the object state/world view for policy designers in cases where time is not a determinant of state configuration or where states are disjoint.

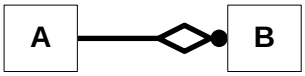


Figure 6: "A implies always in the past B", with its logical interpretation as A(x)⇒ ■B(x) or equivalently as A(x)⇒ ¬♦¬B(x)

However, in our work we are proposing an extensible approach to constraint modeling for relational objects. If a designer's view of the implication (formula) imposed by our interpretation of a graphical construct is different s/he is free to create new constructs. Our purpose in establishing a formal equivalent of a graphical construct (such as a double-headed arrow) is to demonstrate how the modeling language can itself be extended: as any number of new graphical constructs can be introduced provided they translate into well defined logical formulas on the state conditionals. To keep our presentation simple, by not introducing tens and possibly hundreds of different arrows, we restrict ourselves to a few constructs that translate to essential restrictions for business processes. For example, to introduce the temporal constraint specifying "if A(x) is true then always in the past B(x) must also have been true", one could choose the arrow style and interpretation shown in Figure 6.

Appendix B

Performance Analysis and Testing

Observe that in our framework business objects are defined as views that may not be physically materialized. However, maintenance of a materialized object state configuration history, can easily be bootstrapped atop the database auditing framework which itself is often implemented as a set of triggers on base tables. Whether the actual state configuration history is independently materialized or separate components of it are appended to base tables is an implementation level choice. In general when an object changes and an audit log needs to be written, the state configuration at that point in time can be calculated and appended alongside the audit record in the relevant base tables or as an independent materialized view.

There are two sources of overhead that are imposed by maintaining a complete state configuration history. Foremost is the trade-off between space and time, that is, the design decision to store the complete history or to recreate based on the audit logs of objects whenever needed. Our tests show that in most transactional databases in which real-time constraints need to be enforced, the space overhead incurred by materializing a state configuration history alongside the audit trail of an object is negligible. We considered the business definition of a Purchase Order as specified in the TPC-H benchmark to guide our tests. We tested against business scenarios of varying complexity by considering corporate workflow (pertaining to a single object) of size 128 states, 512 states and 1024 states. We utilized TPC-H databases of sizes 1GB and 10GB, on an Intel Core 2 Duo based machine with 4GB RAM running Microsoft SQL Server 2008.

Storage Overhead

Although having 1024 states in a corporate workflow pertaining to a single object is very unlikely, it does represent a plausible upper bound on the number of states that will ever need to be maintained in a state configuration history. Observe that the state configuration history is a very compact representation of an object's membership in all user defined states. Even in the worst case, 1024 true/false results are essentially 1024 bits of information (128 bytes), and this is still less space than a text-based comment field associated with a typical object such as a purchase order (144 character/byte comment field in the TPC-H specifications). Furthermore we believe that since TPC-H is a synthetic performance benchmark, the size of an invoice or line item row in the TPC-H benchmark is an extremely conservative representation of real life business objects and their storage requirements, especially in the presence of large text fields. Consequently having a corporate workflow of 1024 independent states with no conditional variables common in any of them represents an extreme case that will perhaps never be encountered in real life situations.

Our tests showed that in the presence of row level auditing, appending an additional binary field of up-to 1024 bits to the line item audit trail, causes no performance degradation for random updates. Our objective was to test whether a system under a

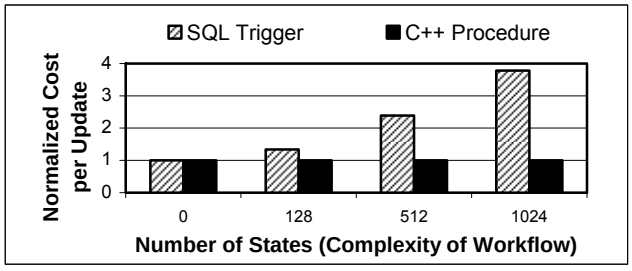


Figure 7: Comparing the computational overhead of a SQL-Trigger based implementation and a C++ implementation

transactional load (high-update situation) is stressed by the additional overhead of writing the state configuration alongside the audit trail. The transactional cost associated with committing an update including the state configuration history to an object

was not significantly different (statistically), from the baseline cost of simply writing the audit log.

The rationale behind there being no additional cost is as follows. A random update to a row in the line item table causes at most two pages to be physically committed to disk, one for the actual modification to the row and another for writing the audit trail in a separate auditing table. As long as the state configuration can be accommodated within the same page as the object, the cost of writing this page will not increase. Thus we argue that, even for the most complex workflows, the state configuration can be stored efficiently and is relatively small compared to the size of the actual object such that there will be no additional storage costs (other than space) incurred by database systems in practical policy situations.

Computational Overhead

For each transaction that modifies an object, our framework requires that the new state configuration of the object be calculated, and then compared against the old state configuration to ensure that all specified constraints are being met. We anticipate that in most business situations where states for business level policy conditions share common variables, we will not have to incur the cost of checking each and every state conditional independently. For example, consider a workflow for invoices with two states called, *paid* and *not_paid*, with the conditions, *paid = true* and *paid = false* respectively. Observe that we only need to check one of these condition to conclude the state configuration for both states. In business situations where a large number of states exist in the policy model, it is very likely that many of them will share the same variables, and thus checking whether an object belongs to several states may be accomplished much more quickly than performing the test for each condition independently. Similarly while the state configuration is being computed we can simultaneously check whether a particular constraint is violated or not and prune the space of constraints that need to be checked dynamically. Nonetheless, in our tests, we took a pessimistic stance on this issue and assumed that there are no avenues of optimization available for us to exploit.

We now summarize the results of the tests conducted to measure the computational overhead of dealing with varying numbers of constraints in a policy model. We noted earlier that there are many mechanisms present in database systems that can be utilized to monitor the implications of an update such as check constraints on base tables, triggers, assertions, and even check constraints on materialized views. Our objective in this section is to provide a reference for comparing the practical computational costs of two of these possible techniques in the light of a varying number of assertions specified in first order temporal logic.

For our tests we utilized various degrees of workflow complexity: 0 states (to represent no computational overhead as a measure of baseline costs), 128 states, 512 states and 1024 states. Adding an extra state essentially means an additional (synthetically designed) check to see whether an object belongs to that state or not in addition to a fixed overhead of checking whether a constraint has been violated or not. These checks were designed to simulate traditional business level string and arithmetic comparisons (such as "*status = paid*", "*shipcode = 'M'* " or "*amount > 0*") that a system will be expected to do to determine an object's presence in each state. These tests were performed sequentially and the results of being present in one state provided no information to determine whether the object will (or will not) be present in any other state.

Not surprisingly the cost per arithmetic/string operation within an SQL trigger is significantly higher when compared to an external more efficiently written program in a language such as C++. Consequently a plain brute force approach of performing a large number of state related checks within an SQL trigger is not the recommended approach for a high performance database system. Figure 7 shows a comparison of the cost incurred when increasing the number of states in a workflow from 0 to 1024. Although most organizational workflows will be in 0-128 state range, it is still less expensive to perform these computations as part of an efficient non-SQL memory resident procedure. In the case of Microsoft SQL Server 2008 this functionality is provided by the ability to execute a pre-compiled DLL (extended stored procedure) and most major database systems offer comparable features for executing external programs. Importantly, when such an approach is adopted there is no discernable increase in the time to test 1024 states than to perform no tests at all.

Appendix C

Maintaining Temporal State Configuration Histories

During the interval between two consecutive committed audit entries with timestamps t_a and t_b, where $t_a < t_b$, an object stored on disk consistently retains the last committed values (of those at t_a). Given that timestamps are of finite length (not a dense domain), it is possible to iterate through all intermediate timestamps and recalculate all intermediate state configurations using the last committed attribute values of an object. Although not very practical, it is possible in general to detect intermediate state changes caused by the passage of time at the time where a new object value is being committed. Therefore we can essentially use a brute force method to solve the problem of phantom state changes. Note that these "hidden" or implicit points of state configuration changes (in our example the point in time when the state of an invoice changes from NEW to NOT NEW) need only be discovered once, and can subsequently be made persistent for efficient monitoring of constraints.

To elaborate the above problem in detail, let us consider an object x that is being updated at time t_b and assume that the last committed state configuration history entry for the object, $SC(x,t)$, was at time t_a. We need to ensure that all intermediate state configuration changes are known to us so that we can enforce the path restrictions specified in the model. Note that for all timestamps i where, $t_a < i < t_{b-1}$, we know that the object was unchanged (still x_a) and therefore we can calculate all intermediate state configurations $SC(x_i,t_i)$ because they will be equivalent to $SC(x_a,t_i)$ and recreate a full extended object history if necessary. However the key points of interested in the range will be where a state configuration change takes place, that is, where $SC(x_a,t_i) \neq SC(x_a,t_{i+1})$.

In practice we do not need to explore the entire temporal state space between two timestamp values to find these points of interest (points where logical state changes took place) but only need to examine the history at the granularity/timeframe of the policy being enforced, which would typically be in the order of days or weeks for most business policies. In fact for most cases,

the time at which the logical state change will occur can be predetermined or scheduled by an analysis of the state conditional specified by the user. The only significant negative consequence of delaying identifying intermediate logical changes in the state configuration until an object is being physically updated is that potential violations will not be found at the earliest possible timestamp.