

Business Policy Modeling and Enforcement in Relational Database Systems

by

Ahmed Ataullah

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2014

©Ahmed Ataullah 2014

I hereby declare that I am the sole author of this thesis. I authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Abstract

Database systems maintain integrity of the stored information by ensuring that modifications to the database comply with constraints designed by the administrators. As the number of users and applications sharing a common database increases, so does the complexity of the set of constraints that originate from higher level business processes. The lack of a systematic mechanism for integrating and reasoning about a diverse set of evolving and potentially interfering policies manifested as database level constraints makes corporate policy management within relational systems a chaotic process.

In this thesis we present a systematic method of mapping a broad set of process centric business policies onto database level constraints. We exploit the observation that the state of a database represents the union of all the states of every ongoing business process and thus establish a bijective relationship between progression in individual business processes and changes in the database state space. We propose graphical notations that are equivalent to integrity constraints specified in linear temporal logic of the past. Furthermore we demonstrate how this notation can accommodate a wide array of workflow patterns, can allow for multiple policy makers to implement their own process centric constraints independently using their own logical policy models, and can model check these constraints within the database system to detect potential conflicting constraints across several different business processes.

A major contribution of this thesis is that it bridges several different areas of research including database systems, temporal logics, model checking, and business workflow/policy management to propose an accessible method of integrating, enforcing, and reasoning about the consequences of process-centric constraints embedded in database systems. As a result, the task of ensuring that a database continuously complies with evolving business rules governed by hundreds of processes, which is traditionally handled by an army of database programmers regularly updating triggers and batch procedures, is made easier, more manageable, and more predictable.

Acknowledgements

I would like to acknowledge the support of my family and friends. The encouragement from my parents, brother, and wife to pursue my passion has been instrumental in my success. I would also like recognize and thank my thesis committee members for their time and effort. Most importantly I would like to thank my supervisor for his input and oversight in my research. Without his guidance and feedback, this work would certainly not have been possible.

Table of Contents

1	Introduction	1
1.1	Background and Motivation	1
1.2	State of the art	2
1.3	The Policy Lifecycle: Inception to Obsolescence	5
1.3.1	Identification and Interpretation	5
1.3.2	Formalizing Policies	7
1.3.3	Implementation	10
1.3.4	Testing, Deploying, and Monitoring	11
1.3.5	Evolution	12
1.4	Problem Statement	12
1.5	Solution Overview	14
1.6	Contributions of the Thesis	15
1.7	Terminology and Definitions	16
1.8	Outline	17
2	Related Work	19
2.1	Outline	19

2.2	Reasoning with Policies	20
2.2.1	Policy Identification	20
2.2.2	Natural Language Processing of Policies	21
2.2.3	Policy Modeling Languages	24
2.2.4	Operational Policy Modelling and Management	28
2.3	Logic, Constraints, and Active Database Systems	39
3	Policy Modelling for Relational Objects	45
3.1	Outline	45
3.2	Premise	46
3.3	Enforcement Model: Moving from Process Models to Constraint Models . .	48
3.3.1	Terminology and Setup	50
3.4	Artifacts in Relational Databases	51
3.4.1	State of an Artifact	54
3.4.2	Restricting the Database State Space	56
3.4.3	Formal Model	56
3.4.4	Starting and Ending Artifact Lifecycles	60
3.5	State Transitions and Database Level Workflows	62
3.5.1	Visual Constraint Notation	62
3.5.2	Artifact Lifecycle Constraints	67
3.5.3	Example	70
3.5.4	Workflow Modelling in RDBMs	71
3.6	Complex Workflow Construction	72
3.6.1	Multi-state Flow Constraints	72

3.6.2	Multiple Paths and Progression	73
3.6.3	Decentralized Workflow Modeling	73
3.6.4	Generalized Path Constraints and Sub-formulas	76
3.6.5	Link between Graphical Constraints and LTL	77
3.7	Related Work	78
3.7.1	<i>Declare</i> Workflow System	78
3.7.2	Integrated Data and Constraint Model	79
3.7.3	Artifact Centric Process Modeling	80
3.8	Summary	81
4	From Specification to Implementation	83
4.1	Framework for Monitoring State Changes	84
4.1.1	Terminology	84
4.1.2	Maintaining Artifact Histories	85
4.1.3	Lifecycle Management	88
4.1.4	Materialization of State Configuration Histories	90
4.1.5	Checking Individual Constraints	92
4.2	Measuring Policy Overhead	92
4.2.1	Storage Overhead	94
4.2.2	Computational Overhead	96
4.3	Optimizations	99
4.3.1	State Space Reduction	99
4.3.2	Constraint Analysis and Sanity Checking	101
4.4	Summary and Related Work	103

4.4.1	Active Database Systems	103
4.4.2	Constraint Systems	104
4.4.3	Summary	105
5	Reasoning over Constraint Systems	107
5.1	Model Construction	108
5.1.1	Terminology	108
5.1.2	The What and Why of Verification	111
5.1.3	Bounded Satisfiability and Model Checking	115
5.2	Database Level Constraint Analysis	121
5.2.1	Defining Conflicts and Errors	123
5.2.2	Cross-artifact Reasoning	123
5.2.3	Making Life Easier for the DBA	128
5.3	Summary and Related Work	131
6	Usability, Limitations and Extensions	133
6.1	On Usability and Complexity	134
6.1.1	Meaningful Constraints over Complex Views	136
6.1.2	Complex Functions and Constraints over Sets	137
6.1.3	A Case for Reduced Complexity	138
6.2	Enhanced Constraint Systems	139
6.2.1	Extending the DCML Toolkit	139
6.2.2	Metadata and Temporal Access Control	140
6.2.3	Metric Temporal Logics	143
6.2.4	Constraint Systems for Auditing Only	145

6.3	Converting Workflows into Constraint Systems	146
6.3.1	Basic Control Flow Patterns	146
6.3.2	Multiple Instance (MI) Patterns	149
6.3.3	A Note on Notation	151
6.4	Summary and Conclusions	152
7	Case Study: Form GC179	155
7.1	Background	155
7.1.1	Purpose of the Study	157
7.1.2	Information Sources and Assumptions	158
7.2	Process Model and Flowchart	160
7.3	Implementation	162
7.3.1	Artifact and State Definitions	162
7.3.2	DCML Diagrams	165
7.3.3	The GC179 Constraint System	173
7.3.4	Verification Results	176
7.3.5	Summary	177
7.4	Will it Work in Practice?	178
7.4.1	The Process	179
7.4.2	Assessing a Policy Management Framework	180
7.5	Summary and Conclusion	185
8	Conclusions and Future Work	187
8.1	Summary and Conclusion	187
8.2	Future Work	189

8.2.1	Probabilistic Reasoning in Database Level Workflows	190
8.2.2	History Mining and Process Discovery	191
A	Zot Model for GC179	193
	References	209

Chapter 1

Introduction

1.1 Background and Motivation

For many businesses, relational database systems (RDBMs) store most of the transactional data involved in day to day operations. An instance of a corporate database can often contain the historic outcome of the activities in which an organization engaged in the past, as well as the current state of individual ongoing business processes. This availability of the entire ‘business state’ has traditionally made database systems the focal point at which operational business policies are monitored and enforced, and RDBMs have accomplished this by maintaining the integrity of the stored information to ensure that modifications to the database comply with constraints designed by the administrators.

However the ever decreasing cost of storage and the widespread proliferation of database systems led to a situation where information pertaining to even the smallest (but not necessarily insignificant) of business events, such as a user clicking on a URL on a web-page or a user checking his/her account balance, is often recorded as a unique row in a database. Although the persistence and availability of ever greater information in a database has significant advantages from a business perspective, it can also lead to significant costs when a business is required to enforce policies relevant to the information being stored. As the number of users and applications sharing this common information increases, so does

the need to make the database system aware of the business logic that governs how data can be accessed and allowed to change over time. In situations such as health-care, government, and finance, the continuous enforcement of complex business rules within database systems is unavoidable because of mandated legislation. It is also in these situations that the requirements of policies are vague and change often. Even within a particular business sector, standardized database deployments or schemas rarely exist, and there are no sets of database level constraints to which every business can adhere. Since each situation is different, the interpretations of policy requirements that manifest themselves as database level constraints are also different for every business.

As a result, database administrators and programmers are left with the manual task to write triggers, check conditions, schedule tasks, and audit procedures that ensure compliance with a given set of rules. In the absence of a systematic method of implementing business policies within a database system, the end result can often be an unmanageable and intricate web of integrity constraints as well as hand written procedures derived from requirements imposed by various operational areas of the business. Implementing database level constraints that enforce complex situation specific rules is non-trivial, and there is a visible shortage of research that attempts to automate this process. This task is largely left to the administrators and consultants specializing in a particular business domain, and the sheer complexity of this problem has spawned several sub-industries of business consultancy that revolve around regulatory compliance in domains such as health-care and finance.

1.2 State of the art

Figure 1.1 presents a generic simplified view of policies pertaining to database systems and how they move from policy text to implementation. The higher levels (policy layer) involve interpreting and bringing together the policies that apply to a business. For large businesses these policies can originate from a multitude of sources, such as taxation authorities, compliance regimes and standards, industry specific consortia, law-enforcement authorities, and private contractual obligations. These policies are typically specified in

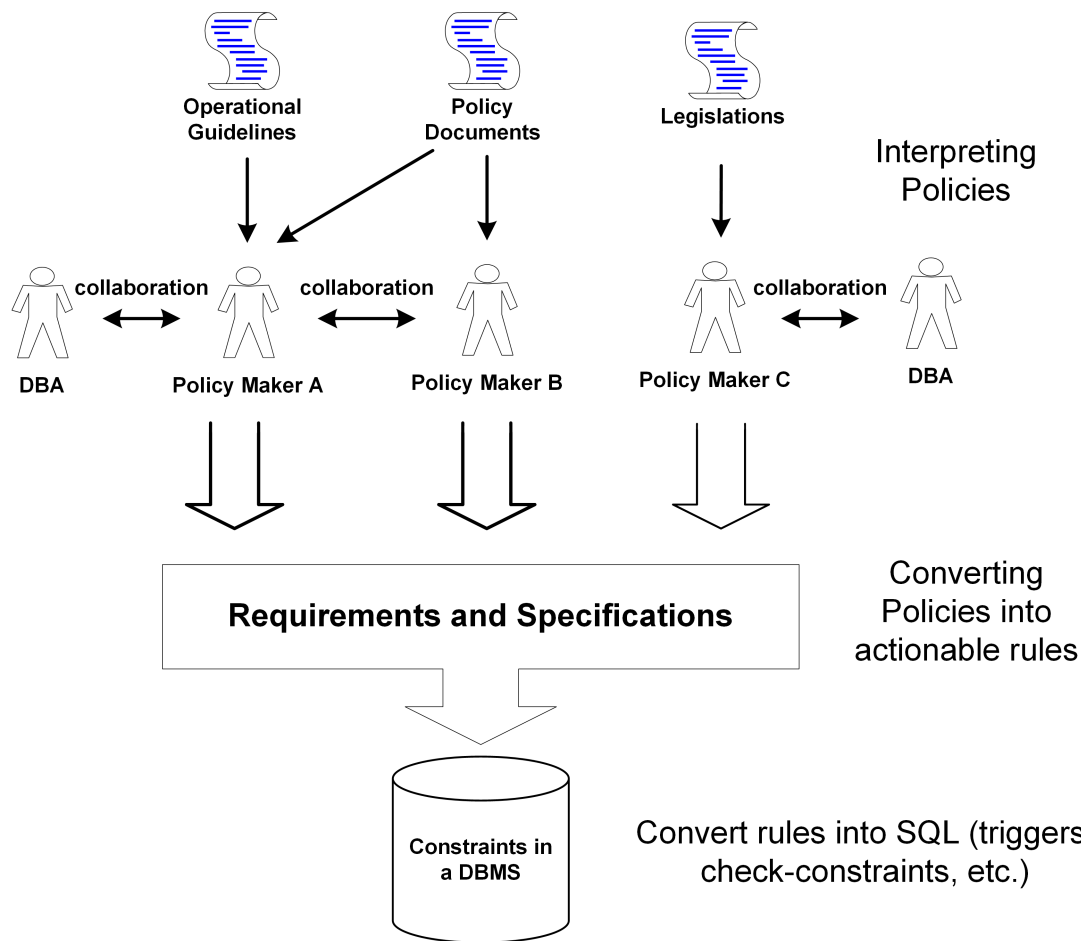


Figure 1.1: High level overview of how business policies are eventually implemented as database level constraints

natural language form for businesses to interpret according to their situation. In this phase questions that need to be addressed by policy administrators almost always pertain to the application rules for the specific business situation: “Does this policy apply to our business? How, where and when is it applicable?”

The intermediate specification layer represents the creation of a formalized (actionable and implementable) interpretation of a policy. At this level, policy rules (when-to-do-what) are made explicit in order to cover the entire space of business situations that an organization can encounter. Policy design and reasoning tools are often used to map out

an easy to understand version of the natural language policies scattered in various source documents. Database administrators may work closely with the policy administrators to decide how best to interpret policies and avoid undesirable consequences when they are eventually programmed as constraints in a DBMS. Note that specifications may not always be well elaborated and developed in all business situations. In fact, it is likely that for small organizations this specification layer is fuzzy (or non-existent) and policies are directly ported into database systems as constraints, without expressing them in an intermediate specifications layer. However for complex businesses, for example situations involving large database deployments with many policy administrators, there is inevitably a need for an intermediate layer, where managers can reason about their proposed policies and examine them in the context of the larger scale business processes. While some organizations may use workflows or flow-charts to describe the sequence of steps to be followed throughout a business and exceptional situations therein, other organizations may prefer to write policies as Event-Condition-Action (ECA) rules. In other words, there are organization specific guidelines governing how specifications are created and exchanged between managers for a particular class of business policies. There is also no universal tool or modelling language that can adequately address the needs of all classes of policies and the preferences of different policy makers. Consequently, there is no standardized technique of communicating specifications to database programmers for implementation.

The lowest level of the diagram (constraint layer) is where the rules contained in the specifications are implemented. This is done by database programmers who write SQL triggers, check-constraints, and batch tasks that accomplish the goals of the policy. At this layer the events, conditions, and actions on business objects contained in a database system are all translated into source code and can be tested against the specifications.

For a concrete example of the above process, let us consider a hypothetical large retail business in Canada implementing the Eco-tax initiative launched by the Government of Ontario [117]. The business must comply with the provincial rules that require end-customers to pay additional fees based on the type of product being purchased. Policy text in this scenario would include not only the actual legislation, but also any supporting material, brochures, press releases, etc. released by the Government of Ontario. Corporate product managers, DBAs, and perhaps even tax lawyers would jointly have to interpret

the word of the law, looking not only at the legislation but also at the product catalog in the database, in order to define the types of products and additional fees that need to be applied. An ambiguity such as “does the eco-tax for tires also apply to bicycle tires,” which is a rule that possibly goes against the spirit of an eco-tax, would need to be resolved before specifications in the form of lists of products, types, and sub-categories subject to the additional fees is derived. Finally the specification will be handed down to the programmers to implement as explicit program code or sets of constraints and dependencies on the types of tax/surcharge rules that can be applied to specific types of products.

1.3 The Policy Lifecycle: Inception to Obsolescence

In order to appreciate the contributions of this thesis, it is important to develop a deeper understanding of the sources of inefficiency in the process of implementing and maintaining an evolving set of policies within a database system (Figure 1.2). We provide a brief overview of the major reasons why the current state of the art of database centric policy management, from the moment it emerges as a business requirement to when it is deemed unnecessary, is costly and can be significantly improved. At each stage in the policy lifecycle, we describe the limits of what automation can accomplish and what aspects of the problem space this thesis aims to address.

1.3.1 Identification and Interpretation

Once a business recognizes that a certain policy needs to be enforced, it needs to identify (or create) sources of information that offer more details and clarifications on the objectives of the policy. This can simply be described as the requirements gathering/analysis part of the policy lifecycle. Policies are interpreted from one or more sources that are often physical documents. Whether it is an internal memo that discusses the needs to have a new check condition in a business process or the enactment of legislation that requires changes to be made in a tax rule in a database, the source (documents, emails, memos, etc.) forms the core on which the implementation will be based.

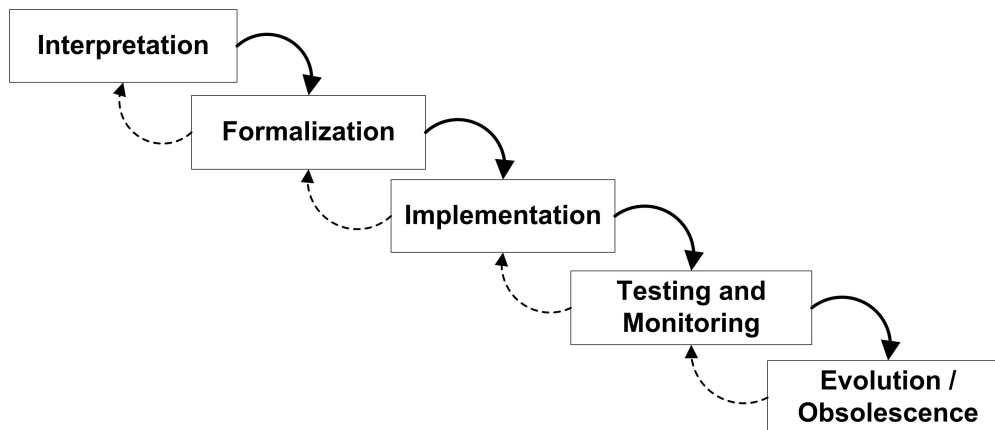


Figure 1.2: A detailed view of the policy-to-constraint translation process. The lifecycle of policies and business rules in database systems very closely resembles the waterfall model of software development

Identifying and interpreting natural language policies and then determining their applicability on specific business operations is a significant challenge within the policy lifecycle. The section of this thesis covering related work (Chapter 2) provides a discussion of automated methods of deriving formal representations of business rules using natural language processing (NLP). The lack of reliability and precision in these automated methods makes them inapplicable in most cases where the implications of misinterpreting policies can be disastrous. Similarly the domain of policy sources is vast and most NLP solutions to interpreting text containing rules remain as research prototypes designed for certain specific sources of policy.

There are many ways in which businesses can adapt to this challenge and reduce the overall cost of policy management. Foremost, an organization can be internally semantically consistent. For example all members of the sales department could have one interpretation of the customer entity. Similarly, the organization could have semantic mappings between the common terms used in various functional areas of a business to eliminate ambiguity among different policy makers. Second, an organization can aim to describe its policies within a common policy modelling framework. That is to say that an organization should try as much as possible to use a single common layer for reasoning about a given class of policies, so that all policy makers can reason with each other’s policies in

a consistent manner. For example at an organizational level the decision to use a single type of process modelling technique to describe its business functions would accomplish this. If multiple workflow or process modelling techniques are employed within an organization, then a mapping between the constructs in each should be established so that cross framework reasoning can be performed at this level before errors are propagated to the implementation layer. Similarly, consistency in the mechanisms for enforcement should also be emphasized at the enforcement level for the same reasons.

Observe that these are merely good practices within the context of policy management, aimed at making interpretation of rules by various policy makers across an organization easier. These are guidelines for interpreters to follow when translating policies from natural language into implementable specifications. The overall problem of extraction of policies from natural language in an automated fashion is intractable. Unsurprisingly this is because of the same reason many of the classical problems in NLP (such as ontology extraction, sense disambiguation, entity resolution, context determination, and topic detection) are intractable: the inherent ambiguity in natural languages. As a consequence, human expertise has always been, and in the foreseeable future will be, required when interpreting policies, rules and laws when applying them to specific situations. Furthermore it is unreasonable to expect that all policies (external to a business) will one day be written under one particular semantic interpretation of the world without any ambiguity and will lead to a single universal interpretation.

The research presented in this thesis is complementary to the problem of automatic rule extraction and interpretation. It focuses on the implementation of rules derived from policy within a database system and assumes that the interpretation, regardless of how it is derived, will be correct. Ideally we see the techniques presented in this thesis to be adopted by policy administrators at all levels, regardless of the methodology they use to interpret policies.

1.3.2 Formalizing Policies

Although not a significant issue for small businesses, it is inevitable that an organization with a complex policy set will have to create an intermediate model for policy rules being

introduced and assess their impact on the overall operations. For example a workflow or a process model could be created by a user to describe the rules by which a specific business process should progress to its conclusion. The task of creating formal models of policy can often be the most costly phase of the policy-to-constraint translation process, and it is important to characterize it precisely. Observe that during interpretation, the objective is largely to identify the scope and sources of policy and data that an organization must examine in further detail to derive a workable specification. However, during this phase the rules are interpreted for a particular system and an actionable set of specifications is generated such that they can be implemented easily. Thus we use the term “formalization” to mean the conversion of policies into actionable specification from relatively informal descriptions.

Consider, for example, the introduction of new data retention legislation with which all businesses must comply. After reading the legislation, a corporate lawyer interprets it to mean that the organization should delete all personally identifiable information of customers with whom the organization has had no dealings in the past three years. Note that the interpretation provides a guideline, but it is not precise enough to be treated as a specification. For example the words, ‘delete’ and ‘dealings’ are vague terms, and many such terms may not be directly related to concepts within a database system. Consequently at this stage (formalization) it is important for managers, DBAs, and corporate lawyers to examine the given interpretation (perhaps all possible interpretations) and to come up with actual actions and safeguards (specifications) that, when eventually implemented, comply with the law.

Mediation and negotiation is often an integral part of achieving a consistent and workable specification for a given policy. Proceeding with the aforementioned example, the simplest implementation of the data retention policy would be to delete all customer entities that are related to transactions that are more than three years old. However this may conflict with requirements from other areas of the business. For example, the marketing department might require that postal codes along with dates and times of customer interactions be preserved indefinitely for research purposes. Similarly, the finance department might need to store and retain purchase records (amounts, taxes, and shipping address of customers) for at least six years in case of a tax audit. Clearly in situations where

shared data is being used by multiple policy makers, there emerges a need to mediate a formal definition of how a new policy should be converted into a specification such that it satisfies all policy makers. There are many reasons why the overall process of arriving at a workable specification is difficult, however the focus of this thesis is on the following two specific problems:

Business Object and Relational Object Mismatch

When business users and policy makers look at policies, rarely do they have a complete understanding of how the objects (nouns and verbs) contained in the policy text correspond to rows in tables that are physically stored in a database system and the intricate relationships that exist between these rows. Even simple tangible concepts, such as an invoice, can be difficult to describe within a database. The contents of an invoice might be spread across multiple tables. An invoice could also be broken down into sub-objects such as line items. In large schemas, complex objects may be scattered across many tables making it much more difficult to pinpoint the relationships between its sub-parts. Eliminating this mismatch between objects referred to in policies and the specific parts of a database that represent those objects becomes extremely difficult when policies refer to aggregate data. For example a health care policy may refer to a *medical history* in several contexts containing various amounts of information, and each interpretation of this phrase needs to be properly identified within a database system. Quite often DBAs have to work closely with policy makers in order to help them narrow down their definition of a policy-relevant object to what actually exists in a database.

This mismatch between a policy maker's world view of objects and the database centric definition of a business object extends to policy actions as well. Policy objectives of seemingly simple business requirements can be accomplished in various ways. For example, instead of 'deleting' a record of customer interactions, parts of that information can be nullified. Equivalently the information can be anonymized; for example, the customer's name and address can be changed to that of a fictitious John Doe in the database. The problem is further exacerbated by the fact that there is no shortage of tools, techniques, and features available in a modern RDBMs that can be used in conjunction to satisfy

the requirements of a given interpretation of a policy. When we couple the mismatch between a single policy maker's definition of a policy object and an object in a database system with similar object-relational mismatches across an organization, we can see that policy formalization and mediation can be a large scale management problem. This is even more so when policy makers use vastly different logical models to express their policy sets. This complexity induced by the sheer number of possible implementations of a policy interpretation makes the DBA a critical part of the policy mediation process.

Conflict Detection and Resolution

Integrating many policies over shared data naturally brings about the possibility of conflicting requirements. Delete-protect conflicts, where one policy maker requires preservation of data and another requires the destruction of the same data, serve as ideal examples of this problem. Quite often these conflicts will slip through the modelling and reasoning phase, either because of a lack of meticulous oversight or the inability to reason across multiple logical policy models.

1.3.3 Implementation

Once a specification of policies is sufficiently well-defined, it is handed over to database application programmers for implementation. Their role is to translate the specifications using the database framework of an organization into actions and safeguards that accomplish the goals of the policy being introduced. Database administrators and programmers need to be aware of the best possible tools and techniques that are available within their corporate RDBMs to accomplish this translation. In a situation where numerous policy changes are being introduced at the same time in a database system, it may be beneficial to chalk out a plan for implementing the new policy requirements. For example, if the physical schema of the database needs to be modified, then the implications of the change (such as the impact on logical views) need to be examined carefully. In complex systems this planning phase will certainly be a costly and time consuming process. The core implementation cost (programming time) associated with writing triggers, check constraints,

and stored procedure that need to be executed periodically to audit the policy can also be significant. Since the upper layers of interpreting and formalizing the policy are largely manual processes, there is little scope for automating the implementation phase.

The lifecycle of a database specific policy (Figure 1.2) has the property that at any particular phase a policy can be pushed back to the upper layers for re-examination. Consider a policy from a corporate security officer that requires access requests issued to a database server to be logged for privacy audits. During implementation it is discovered by programmers that the continuous logging of queries greatly slows down the database server, which impedes day to day operations. A management decision at this point could possibly be to re-interpret the policy requirement instead of dedicating more resources for the database server. For example, the auditor may accept a less data intensive record of login and logout times as sufficient information for his/her purposes rather than requiring a full log of queries issued by each user. Consequently, the security officer may be asked to modify the specifications by possibly re-examining and re-interpreting the source documents pertaining to the policy.

1.3.4 Testing, Deploying, and Monitoring

Systematic testing and deployment of new database level rules is very similar to deploying a typical software update. Since some policy conflicts within a database system can only be detected at runtime, the purpose of testing is to detect specific conditions under which the new policy rules will adversely impact the system. As part of deploying resilient policies, it is often the case that routines that audit the policies are written independently thereby increasing the cost of a policy implementation. For example, while one programmer may implement a mechanism to purge complex records containing private information, a different programmer may be given the task to write the auditing procedure that verifies the non-existence of such private information after the original procedure is executed.

Once a policy set is activated, its associated rules and obligations pose a long term ongoing cost in terms of monitoring. Conflicts and warnings generated by policy rules may need to be reported, and, depending on the larger business process, various actions may need to be taken. Automation among these actions, although difficult to verify, is

relatively easy to implement. Typical examples of these long term monitoring actions include emailing relevant parties with the details and the findings of the execution of the policy rule.

1.3.5 Evolution

There are two main aspects to changes in a stable policy set that every business must consider. Foremost an organization may encounter scenarios where amendments to a policy may manifest themselves over time. An implementation that is complex invariably requires more time to modify and thus costs more to maintain. Since the scope of the change in policies cannot usually be anticipated, even with well-implemented policy sets there is always the risk of running into a simple change in requirements that can be costly to implement. Second, we note that corporate databases, like businesses themselves, are also constantly evolving and subject to new constraints that originate from operational requirements. For example, a database schema may change as a result of a new application being developed to support other business requirements. Similarly, new sources of data that get integrated within a database system may enhance the scope of older policies, and thus they may require the entire policy lifecycle to be repeated for specific policies. Again the bottom line is that reducing the ongoing cost of maintaining a complex policy set, using a systematic way of managing policies within database systems, represents a significant opportunity for businesses.

1.4 Problem Statement

The envelope of business level policies implemented today within database systems has vastly expanded from classical integrity and access control constraints to include auditing, usage control, privacy management, and records retention. Traditional integrity constraints, for example, asserting that account balances cannot be negative or that ZIP codes, email addresses, and phone numbers are in a certain format, are essentially static invariants, and their governing rules rarely change. On the other hand, process centric policies

change continually with business requirements. Temporary changes in business processes, for example a policy that requires that “for the next two months Jane can only pay an invoice if it has been approved by John in the Accounting Department,” typically pose more significant and ongoing costs. Similarly complex temporal constraints, for example, one that requires that “an employee’s salary cannot increase by over 20% in any two increments,” are much harder for business users to formalize and equally hard to enforce in a traditional database system that may require changes to the database schema to support such policies. As users become more aware of the social and business implications of database technology, the amount, type, and complexity of policies being implemented within database systems is likely to increase. However, little research has been done in the database community to address the absence of a systematic methodology for reasoning about database specific policies, and it remains a substantial area where improvement is required.

There are major shortcomings in the current state of the art in converting business rules into database level constraints. Foremost, it is impractical for different classes of users including legal experts, business managers and policy makers, database administrators, and database programmers to communicate and reason over the same policy constructs. Consequently, policy interpretation, formalization (conversion into specifications) and implementation has to be done by three very different user groups. Conflicts across these three layers are difficult to detect and costly to isolate, and repairs involving re-interpretation and mediation can take a long time. The process to convert policy rules to database level constraints is labour intensive, thus making total compliance at the database level a costly proposition. Finally, because of the ad-hoc way in which policies and constraints bubble down to the database from different areas of a business, verification of these constraints and producing compliance guarantees is infeasible.

A major avenue of improvement that prior work has not examined is that of empowering business level users and policy makers to develop database level constraints themselves. A significant hurdle in doing so is that policy makers specializing in various business domains are not database programmers and thus not capable of writing complex queries and event-condition-action rules that implement their envisioned policies. Consequently, they have to rely on database programmers to convert specifications into implementation.

1.5 Solution Overview

This thesis presents a method of translating informal operational specifications of business processes, such as those contained in workflows and process models, into database level integrity constraints. The fundamental observation exploited in this work is that databases and business are essentially equivalent state machines and policies can be reduced to directives on how these state machines should behave. Just as a sequence of database transactions can lead it to an inconsistent state with regards to a policy, certain paths that a business traverses are invalid with regards to compliance with the same policy. It is argued that a mapping between equivalent paths in these state machines is easy to establish, even by business users who are not familiar with database terminology. Furthermore, there exist many ways to create, examine, and reason about these mappings under different policy scenarios, such as those encountered in business process modelling, access control, obligations management, records retention and disclosure control. Most importantly, in each of the above scenarios a policy maker can use his or her favorite visual model and modelling tool to engineer and create these state space restrictions without worrying about the ramifications of models created by other policy makers. In essence, the need to have a DBA acting as a central mediator of policy is virtually eliminated, because policies can be directly created by the responsible parties in a distributed fashion.

This thesis also demonstrates that once this formal mapping between paths in the business state space and database state space has been established, automatic translation of policies into database enforcement actions can be done effortlessly. For a large class of policies in the aforementioned policy domains, conflicts can also be detected statically, and proofs of compliance (showing that inter-policy conflicts will never occur) can be generated by reasoning over the path constraints specified in the policy model. As a consequence, the middle stages of the policy lifecycle are greatly simplified and the overall cost associated with managing a large policy set over a database system is significantly reduced.

The above description of the solution can be summarized in the following thesis statement:

Thesis Statement: Businesses and databases can both be viewed as corresponding state machines, and it is feasible to translate the behaviour of a business, as specified in individual high-level business process specifications, into database level integrity constraints. Furthermore, the correspondence between specific parts of a database and a business process can be established by non-database experts, used to generate the relevant integrity constraints, and exploited to formally reason about the behaviour of several processes interacting over shared data.

1.6 Contributions of the Thesis

The thesis has two major contributions. First it presents a modeling language for implementing a large class of business level rules as database level temporal integrity constraints. The specific properties of the language that make it a significant improvement over existing techniques are as follows:

1. It is highly customizable and applicable in a broad range of scenarios for which traditional process modeling techniques are inadequate (e.g. temporal access control). The language extends the state of the art in workflows and business process modeling and provides much richer constructs for constraint specification over business objects contained in database systems.
2. The language resembles business workflows and process models, and thus it is easy to use for business policy makers. It can also serve as a coherent, unified policy specification layer for a broad class of business objects contained in database systems.
3. The modeling language serves as a specification language and can be automatically translated into database level temporal integrity constraints. Consequently, constraints specified by users can be directly implemented without any DBA assistance or manual programming.

Second, the thesis introduces and examines the notion of model checking business processes within relational database systems. The thesis is the first to attempt to verify the

behaviour of the contents of a database in order to make claims about the process that the changing database represents. Connections between the modelling language presented and how model interactions can be examined are discussed in thesis. Limits of verification, such as enforceability of constraints, formulating properties for verification, and issues pertaining to efficiency in verification, are examined. Lastly the role of the DBA in formulating properties that need verification is discussed, and the implications of conflicting policies over shared data is examined.

1.7 Terminology and Definitions

The following is a list of terms along with their meaning that are used throughout this thesis. Every effort is made in this thesis to use these words and phrases in a consistent manner as specified below:

Policies and Constraints : This thesis uses the word *policy* in a informal fashion to mean a set of rules applicable to a business. Policies in the most general sense are guidelines established to achieve a specified outcome. For example, “access to data should be restricted to ensure confidentiality and privacy” is a broad policy statement. Policies are also implementation agnostic. In contrast, the term *constraint* is used in this thesis in the classical database theoretic sense, as an integrity constraint or a data validation check condition within a database system. The notion that these two can be related (via a correspondence) is examined in Chapter 3.

Workflow or (Business) Process Model : Although from a business perspective these two terms are differentiated by the granularity in which they specify a repeatable task (a process often being a more specific notion than a workflow), the reader can consider them to be interchangeable for the purpose of this thesis. With regards to the work presented, a *workflow* or a *process model* is simply a sequence of steps requiring that a particular class of objects complies with the mandated business level policies during its lifecycle.

Auditing : The term auditing with regards to database systems is used to assert that a log or a trace of events/actions/updates is being maintained. Consequently, in the database theoretic sense the term auditing and logging are used interchangeably. The thesis differentiates this meaning of auditing in databases from its meaning in business and finance, which is interpreted as ensuring a business complies with some published policy.

1.8 Outline

The remainder of this thesis is organized as follows. Chapter 2 examines the related work pertaining to policy management (interpretation, specification, and implementation). In Chapter 3 we introduce the Database Constraint Modeling Language (DCML), our visual notation for denoting process-centric constraints in database systems. We present the semantics and demonstrate, using examples, how informally specified business process descriptions (such workflows and process models) can be translated into DCML models over business artifacts contained in a relational database. In Chapter 4 we examine how DCML models can be implemented as active constraints that monitor and maintain the integrity of a database with regards to the constraints embedded in them. Issues pertaining to efficient monitoring of constraints derived from DCML models are examined therein.

Chapter 5 introduces the notion of model checking individual and sets of related DCML models. A mechanism for identifying conflicting constraints across models is presented, and the role of the DBA in formulating properties about DCML models that need formal verification is introduced. In Chapter 6 we critically examine the usability of DCML in practical situations. We also discuss how DCML could be extended and tailored for specific applications (types of policies and policy makers) and the limitations imposed by the design choices available to policy makers. The capabilities of DCML are compared against traditional workflow modeling languages in this chapter as well. Finally, Chapter 7 concludes this thesis by presenting a case study detailing how DCML could be used to constrain a database for a real world business process. It discusses avenues of future work and presents our proposed methodology with which a business can measure and benchmark

the efficacy of DCML and the associated verification infrastructure against the state of the art.

Chapter 2

Related Work

2.1 Outline

The contents of this thesis relate to a large amount of research literature in several areas of policy management and computer science. Consequently this chapter provides a brief review of the related research throughout the phases of the policy lifecycle. We introduce terminology used in prior work that is also employed in subsequent chapters and develop the groundwork for appreciating the research presented in this thesis. First, issues pertaining to the translation, interpretation, modeling, and reasoning of policies in the abstract are discussed. Then research related to the conversion of policies into database level constraints is presented. Finally, prominent auxiliary problems such as the testing, verification, evolution, maintenance, and obsolescence of constraints embedded within database systems are presented.

In addition, each subsequent chapter in this thesis is followed by a smaller more precise related works section that compares and contrasts the specific contributions of this thesis to prior research. The objective in doing so is to avoid overwhelming the reader with a large amount of research at different levels of policy management that is complementary to the contributions of this thesis yet essential to appreciate the avenues of future research opened by it (Chapter 7). At the same time this organization presents our contributions

in such a way that they can be easily compared and contrasted with the specific aspects of prior work.

As mentioned earlier, policies in their broader meaning are simply guidelines established to achieve specified business objectives. However, within the context of constraints embedded in database systems, this thesis classifies constraints according to how they accomplish one or both the objectives of (i) a security policy or (ii) a process-centric policy. Access control, privacy management, and data retention are classical examples of security policies, whereas integrity constraints on data, for instance the requirement that the customer account balance attribute must always be positive, are manifestations of operational, process-centric policies. The remainder of this chapter is organized around examining past research on policy management (different aspects of the policy lifecycle) across these two broad policy objectives. As we go from the start of the policy lifecycle to the end, we examine research contributions that address the challenges associated with the particular stage of the policy lifecycle across several subclasses of security and operational business policies.

2.2 Reasoning with Policies

2.2.1 Policy Identification

Identifying sources of policy is primarily not a computer science problem and is most closely related to the study of corporate *governance, risk management, and compliance* (GRC). Even though GRC encompasses issues related to the management of corporate information systems, the specific problem of identifying policies is more a legal art than a technical challenge [128]. Several recent papers [11, 20, 32, 81] and books [151, 96, 143, 78] on GRC have examined the implications of changes in the legal landscape (such as the introduction of HIPAA [1] and the Sarbanes-Oxley Act [2]) for corporate information systems. There is indeed growing concern among businesses to be proactive when it comes to legal compliance and ensuring that risks associated with the ownership and safekeeping of personal information are minimized. However, policy identification and topics addressing

how, when and, where policies (and associated non-compliance risks) emerge are clearly beyond the scope of this thesis. The reader is directed to the above sources for further reading on these matters. For the remainder of this chapter, we will not question the sources of policies. Instead we will focus on the way these pre-identified policies are specified and interpreted for implementation.

Policy specifications, much like software specifications, can be categorized into three types based on how closely they resemble their implementation. At one extreme we have policies in natural language, which are usually written in free form text for human consumption. At the other extreme is the implementation, which, in a manner of speaking, can be considered a policy specification albeit only readable and interpretable by a knowledgeable programmer. In the middle lie a slew of modeling languages for rule analysis that attempt to bridge the gap between expressivity and actionability for specific situations. As an example in the domain of security policies, consider a privacy statement of practices that will be typically written and edited by an administrator or a manager. This document may also be available on a website for clients and other stakeholders to view. Since such a document will be broad ranging and govern the overall behaviour of an organization, other more specific documents may be derived from it to cater to specific corporate systems such as the email server, corporate document management system, and corporate database system. Then, these specific documents are converted into rules for different computer systems, possibly via an intermediate language such as P3P (Platform for Privacy Preferences). The specifications in these intermediate documents are eventually handed down as constraints for programmers to implement in their specific systems across the organization.

2.2.2 Natural Language Processing of Policies

Since interpreting legal documents is an inherently ambiguous and error prone process that most software engineers are unable to do well [104], one can be misled into believing that this task can be better accomplished using automated methods. However NLP based systems that can independently generate perfect models of rule systems from policy text are currently far from being a reality [27].

There are several noteworthy papers that have attempted to create structured models of rules contained in text. For example, Moulin and Rousseau [106, 105] present a syntactic parser that attempts to extract deontic rules from legal text. Their approach is specially geared towards *prescriptive* text, which either explicitly allows or disallows actions to occur. A similar prolog based rule parser and a proof-of-concept was discussed in the work of Michael, Ong and Rowe [103] and in Ong’s thesis [116]. The results of both these works were encouraging; however the lack of broad and rigorous testing across different types of text leaves much work to be done in these parsing approaches. In contrast, the research of Xiao et al. [174] takes a much more focused approach. It is largely motivated by the fact that access control requirements are typically not presented as a separate document to software developers. Instead access control restrictions are often embedded and scattered across hundreds of pages of software specifications, making it very difficult to capture the security specifications of a system during software development. Their approach uses linguistic analysis in order to identify sentences that relate to access control requirements contained in a specification document. The claimed accuracy of their proposed sentence level classification technique ranges between 80% and 90%. Since their goal was to only identify sentences that relate to access control requirements, the task of building a model for these requirements is unresolved.

It is generally accepted that a hybrid or human assisted approach to extracting various policy models is perhaps a much more realistic and effective way to convert natural language policies into formal models. The most prominent such approaches are those of Kiyavitskaya et al. [92, 91] and Breaux et al. [28, 27]. Kiyavitskaya and Breaux, together with several other collaborators, have developed a system that allows for user-guided construction of a policy model through annotation and identification of subjects, roles, and obligations. The user is expected to guide the system through the policy corpus interactively as it builds a formal model of the rules contained therein. Arguably this technique significantly reduces the chances of misinterpretation and missing policies within text, as human oversight is ever present.

While user-assisted systems may perform better than either statistical or syntactic approaches, their adoption has been lackluster. This is most likely because of the inapplicability of the proposed research prototypes to real life situations, as well as incompatibility

with access control mechanisms deployed in existing systems. One such project that not only embraces the need for human interaction in policy interpretation, but also attempts to make it a useful exercise for the user, is SPARCLE [29] (not to be confused with the SPARQL query language). The objective of SPARCLE is to allow user-assisted parsing of policy documents with the objective of mapping them to existing (or soon to be implemented) constraints. In the future, developers of SPARCLE hope to allow policies authored within the system to be plugged into existing policy oriented software. However the currently published research associated with this project [29, 131, 90, 165] has focused on HCI and evaluating the usability of semi-automated policy parsing/authoring tools.

Other work that has examined problems associated with policy authoring with little or no assisted policy parsing include that of Shi and Chadwick, who presented a system for authoring restricted class of access control policies [141], and De Coi et al., who examined the authoring of privacy policies for social networks [39]. The former approach is especially interesting, since it offers an insight into how a controlled natural language, a subset of a natural language limited in expressivity so that it is easier for formal reasoning, can offer an ideal compromise in this problem area. Our work acknowledges this overall compromise in expressivity and usability when it comes to automated policy systems, and a significant portion of Chapter 6 is dedicated to this discussion. The complementary (perhaps the logical inverse) problem of generating human readable policies from a set of formal rules has been examined in the context of network security policies [40]. NLP techniques have also been employed in examining process-centric policies. For example, the work of Friedrich et al. [66] looks at the problem of constructing business process models from their natural language descriptions. Although the technique presented relies on sentence level parsing, it attempts to capture the links between the sentences in order to establish continuity for identifying sequential and conditional actions in a business process. After testing their technique on 47 text-model pairs from industry and textbooks, Friedrich et al. claim accuracy of 77% for their proposed technique. Similar approaches to extract and construct UML and use-case diagrams from text have also been studied [145, 146].

It is important that we acknowledge the vast amount of literature pertaining to NLP in general (i.e., not specific to text containing natural language policies). Several problems in this area, such as ontology extraction, entity recognition, and sense disambiguation, are

directly applicable to parsing of business policies. Common themes in the prior work on processing business policies for rule extraction include the use of a shallow parser or weak grammars in a template based syntactic/linguistic technique to identify business rules in policy text. Such techniques take a very simplified view of the corpus and therefore lack the ability to offer a comprehensive policy interpretation framework for complex rules such as those governed by temporal operators or a deeper understanding of the problem space.

2.2.3 Policy Modeling Languages

Modeling of policies and processes has two well accepted objectives [46, 148]: (a) Summarization or simplification of policy text into a set of requirements that is easier to consume than the source documents and (b) formalization, through which it becomes possible to reason about policies and eventually implement them within various systems of an organization. Consequently, the most well cited surveys in the areas of policy modeling [100, 3, 148] have attempted to differentiate modeling languages across these two dimensions. Modeling techniques that are graphical in nature, such as workflows and Petri nets, often emphasize clarity and simplicity, whereas models that are logic or rule based such as event-condition-action lists focus on achieving the benefits of formalization. In many cases a graphical model can have an equivalent logical interpretation and vice versa, offering a compromise between formalization and simplification. Many modeling techniques have been proposed in the prior literature, and in this section we provide a brief overview of the most popular and prevalent ones.

Privacy

The Platform for Privacy Preferences (P3P) [44, 130, 4] is W3C's privacy preference specification language, which allows web based exchange of privacy specifications among users (agents) and different web services. Using P3P, end users can limit the type of information they wish to communicate with web sites and only do so if the web site promises to use it for specific purposes. Several popular browsers today, such as Microsoft Internet Explorer, have built in support for exchanging P3P policies with web servers [41]. According to

surveys, roughly 15% of the top 5000 visited web sites implement a mechanism for accommodating end user P3P policies [156]. Unfortunately since P3P policies are not considered legal contracts [132], businesses have a very high incentive in over promising or not keeping promises at all when it comes to acknowledged user privacy preferences.

Enterprise Privacy Authorization Language (EPAL) [13, 134] was a similar attempt at specifying and exchanging privacy preferences, at a much finer granularity than P3P. However EPAL was never formally adopted as an OASIS (Organization for the Advancement of Structured Information Standards) standard. EPAL was eventually superseded by eXtensible Access Control Markup Language (XACML), which, like its predecessor, is a declarative specification language for access control policies accompanied by a processing model that specifies how policies are interpreted.

Security and Access Control

Today XACML has been widely accepted as the only comprehensive standard that accomplishes the objectives of EPAL, P3P, and several other access control specification languages that were proposed alongside these. In fact, due to its flexible, XML-like syntax, several extensions of XACML are now themselves standardized languages for exchanging specifications. For a comprehensive comparison of XACML with prior languages, the reader is directed to the work of Anderson [10].

Even though XACML accomplished goals well beyond those for which it was initially proposed and represents the state of the art in access control policy specification, it has several shortcomings. Foremost, XACML is purely a specification language and compromises heavily on formalization for simplicity (see Listing 2.1 on page 27 for a simple example of a policy in XACML). Furthermore, XACML does not go far beyond providing an easy to use specifications framework for attribute based decision making. A typical XACML policy will specify subjects, actions, objects, and conditions under which actions should be allowed or not allowed, but the interpretation and implementation of these policies in various systems remains an open problem for the developers of policy vocabulary and developers of the specific systems. This is perhaps why the database research community has been impacted very little by the increasing popularity of XACML. The fact that the subjects,

actions, and objects mentioned in an XACML policy still need to be explicitly mapped and translated onto individual pieces of data and constraints (at the database level using traditional grant/deny statements) makes XACML relatively ineffective for conveying any useful information to a database programmer. Recently there have been attempts made within the database community, more specifically by Jahid et. al [82, 83], encouraging database security administrators to encode database level constraints using XACML in order to exchange and recompile them efficiently across various systems. However they do not address the question of how to derive database constraints from specifications in the first place.

There are, of course, several other policy modeling paradigms, such as Role-Based Access Control (RBAC) [64, 63, 138] and its temporal extension [25, 88], Label-Based Access Control (also known as Lattice-Based Access Control or Rule-Based Access Control) [137, 54], and the classical ACL-Based Discretionary Access Control [139, 58]. A major shortcoming of the policy modeling techniques in each these paradigms is that they limit modeling to a few core attributes, such as subject, target, condition, and action. As we shall see shortly, very few access control policy modeling techniques focus on the set of business processes and specific situations under which a policy decision is to be made. The classical motivating example for such flexibility in modeling access control comes from the domain of health care, where a process (such as handling a medical emergency) may require that general access control restrictions need to be ignored for specific cases in order to best serve a patient. Recent proposals, such as Usage-Based Access Control [136, 175], and Task-Based Access Control [110] have attempted to address this divide by integrating roles and situation specific attributes together. Unfortunately these proposals only marginally extend prior approaches with additional attributes and modes of reasoning over situation specific attributes. The general problems of (a) conveying access control constraints as part of a business process and (b) mapping these modelled constraints into various systems that implement them are not addressed in these works.


```

1
2 <Policy PolicyId="p1" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:permit-overrides">
3   <Target>
4     <Subjects><AnySubject/></Subjects>
5     <Resources><AnyResource/></Resources>
6     <Actions><AnyAction/></Actions>
7   </Target>
8   <Rule RuleId="Rule1" Effect="Permit">
9     <Target>
10      <Subjects>
11        <AnySubject/>
12      </Subjects>
13      <Resources>
14        <Resource>
15          <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
16            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
17              StudentList
18            </AttributeValue>
19            <ResourceAttributeDesignator
20              AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
21              DataType="http://www.w3.org/2001/XMLSchema#string" />
22          </ResourceMatch>
23        </Resource>
24      </Resources>
25      <Actions>
26        <Action>
27          <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
28            <AttributeValue
29              DataType="http://www.w3.org/2001/XMLSchema#string">read</AttributeValue>
30            <ActionAttributeDesignator
31              AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
32              DataType="http://www.w3.org/2001/XMLSchema#string" />
33          </ActionMatch>
34        </Action>
35      </Actions>
36    </Target>
37  </Rule>
38 </Policy>

```

Listing 2.1: An example of a simple XACML Policy, taken from the Master’s thesis of Jin [86], explicitly allows anyone to read (action) the Student List (object)

Data Retention

Data retention is a subfield of records management that deals with the proper safekeeping and destruction of data according to a scheduled and pre-determined lifecycle [60, 61]. Most businesses today have legal requirements to safeguard their operational records, such as those used for tax purposes, for a specified number of years. Furthermore many organizations, such as health care providers, are required by law to remove records from their system after a certain period of time has elapsed in order to protect the privacy of patients. Similar legal requirements, when combined with the facts that maintaining a large amount of sensitive information can be a liability and that possession of certain information such as incriminating emails can lead to severe losses, has led to a situation where businesses have started to take a keen interest in formulating and enforcing data retention policies.

The state of the art in specifying retention requirements is essentially attribute-based retention constraints on classes of business objects. For example, in the domain of document management, each document in a corporate enterprise content management system (ECM) could have a set retention period based on its sensitivity label. Systems such as OpenText’s LiveLink ECM can use these labels and conditions such as “a document not being accessed for 7 years” to automatically remove them from the system [43]. Similarly, policies on emails, such as those present in typical corporate email storage servers (e.g. Microsoft Exchange Server [42]), are often enforced transparently without users being aware of them. An example of such a policy could be that emails that are processed/accessed today will be deleted in 365 days unless labelled as important. We have extensively examined the notion of what is a record in a database and the implications of protecting and deleting different types of records stored within relational database systems [15].

2.2.4 Operational Policy Modelling and Management

The term *process modelling* refers to the holistic study of everyday repeatable business functions. The hundreds of process modelling techniques, ranging from classical workflows and flowcharts to current day extensions of UML diagrams, offer a varying degree of compromise between ease of comprehension and ease of implementation. The process

modelling languages that are purely abstract do not contain sufficient detail to delineate exactly how a process should (or should not) progress further towards its conclusion. In most cases, the objective of these higher level modeling techniques is to provide a clear, concise, and simplified graphical (or pictorial) snapshot of a complex multi-stage process so that managers can analyze it and make improvements. Traditional wisdom suggests that models created by these techniques are valuable because a non-specialist (such as administrative staff or an external consultant) can in a few seconds gather a large amount of high level information about processes. Moving to the other side of the spectrum, one can observe that the more concrete process modelling techniques are often workflow oriented, and they may even specify pre and post conditions for individual business objects that need to be met as a means of restricting control flow of objects and processes. The aim of many of these flow oriented languages is to provide deeper insight into processes and their sub-stages, so that these models can be provided to system developers. However, the details of the implementation, for example, how each pre/post condition is mapped onto the implementation level (for example, the specific pre-condition on business objects stored in a database system), are often not specified. As we move ever closer to implementation, there can exist implementation specific modeling techniques. Common examples of modeling techniques that can accomplish automatic model-to-code translation include the large number of widely used CASE (Computer Aided Software Engineering Tools), ER-modelling, and UML-to-code extensions for common IDEs, such as IBM Eclipse. These tools have traditionally generated method stubs for programming languages (including SQL) based on business level models but can even go as far as generating specific branches and assertions in program control flow to ensure that the code that is eventually written is true to the model. The focus of these techniques has remained at source code generation for programs, rather than database level constraint generation and maintenance.

There is no shortage of process modelling languages and workflow diagram semantics available today. Several books and surveys have been published, examining the myriad of very similar techniques for workflow modeling. The interested reader is directed towards the work of Fischer [65] and van der Aalst et al. [164, 162] as comprehensive sources that examine workflows and towards the work of Ko et al. [93] for a more recent survey that attempts to categorize and differentiate many of the proposed workflow modeling

techniques. In the process modelling community there are also numerous surveys and books that provide insight not only on the wide array of notation and techniques [76, 7, 69, 99], but also on best practices in the industry [21, 180, 85, 102]. Because of the sheer number of modeling languages available, the usability and popularity of each one is hard to gauge. However, the most well cited ones in the literature are invariably graphical and also have logical interpretations to them. It is very likely because of the lack of a easy-to-visualize graphical representation that pure logic/functional descriptions of business processes are not popular among business users. The seemingly simple idea of van der Aalst et al. to characterize different workflow techniques based on a set of common workflow patterns [163] has had a prolific impact in this field. Workflow patterns are basically design patterns that attempt to classify various types of control flow dependencies in workflows. Examples of these dependencies include *sequence* (a task must happen before the other), *parallel split* (execute multiple tasks in parallel in any order), and *synchronize* (wait for all tasks to complete). Although there are other classifications, the seminal work of van der Aalst proposed 20 such patterns, which are now the gold standard against which different workflow languages are compared.

In this section we examine only classical approaches to process modelling that have had impact on the database community when it comes to implementing database level constraints and assertions. This examination of well understood approaches will lay down the groundwork for examining specific properties of our proposed method of modelling database level constraints. To concretize and differentiate the contributions of this thesis further, at the end of Chapters 3 and 4 we will take a closer look at more recent approaches in database specific policy modeling and those that more closely share specific characteristics with our proposed technique. In Chapter 6, in order to show that the expressive power of our proposed modeling technique is broad enough to capture most workflow situations, we will also revisit the notion of common workflow patterns as introduced by van der Aalst et al. and examine how our modeling technique provides facilities for implementing these patterns.

Entity Relationship Modeling and its extensions

By far the most prolific database modeling technique of today is entity-relationship (ER) modeling [140]. Although traditional ER diagrams do not capture the deeper procedural interactions within a business process, they do capture data-centric definitions of business objects and their relationships to each other. In addition to fleshing out exactly how business objects will be stored within relational database, ER diagrams provide some very limited support for the designer to enforce some basic schematic constraint on the logical structure of the database. These constraints include cardinality and referential integrity which, for example, can be used to specify that a business object can/should be related to at most/least some number of other business objects through a certain relationship.

By the late 90s the overwhelming popularity of ER modeling had prompted researchers to develop a myriad of temporal extensions to ER modeling. Because the original ER model and its extension (Enhanced Entity Relationship Model [153]) described in practically all modern database textbooks leave time as an implicit attribute of entities and relationships, it becomes impossible to reason about time sensitive relationship and temporal constraints in ER modeling. This represents a significant problem when designing databases that store historical or time-annotated data, where the valid time of a fact (the time periods in the past, present, or future where a fact remains true) are fundamentally different from its transaction time (the time at which the fact was recorded in a database). An example of this complexity arising from different transaction and valid times can be given in the context of a seemingly simple data item, namely marital status. Depending on the interpretation, this attribute can have complex temporal intricacies and constraints associated with it. For example, every individual must start off as “single” and after their expected date of marriage is known in a database system, their status from that point in the future will change to married. If the individual files papers for a divorce with a future effective date in the application, only from that point onwards will his/her status change to “divorced.” Similarly there can be inter-related rules and constraints between concepts. For example once married, a person can never revert to single and must become either divorced or widowed. In the case a person marries twice but the second spouse subsequently passes away, precedence rules based on temporality may be applicable to determine which status

(widowed or divorced) is implied by the sequence of events.

The most recent and well cited survey in the general field of temporal data management was published in 1999 by Jensen and Snodgrass [84]. It concluded that the dozen or so temporal entity relationship modeling techniques that had been proposed by 1999 had seen little to no industrial adoption. In the same year, Gregersen and Jensen published a much more comprehensive survey comparing various temporal entity relationship modeling techniques [70]. The following paragraph taken directly from that survey, still holds true today, and eloquently summarizes the shortcomings of temporal ER modeling techniques:

“In the research community, as well as in industry, it has been recognized that temporal aspects of database schemas are both prominent and difficult to capture using the ER model. Put simply, when modeling fully the temporal aspects, the temporal aspects tend to obscure and clutter otherwise intuitive and easy-to-comprehend diagrams. As a result, some industrial users simply choose to ignore all temporal aspects in their ER diagrams and supplement the diagrams with phrases such as “full temporal support.” The result is that the mapping of ER diagrams to relational tables must be performed by hand; and the ER diagrams do not document well the temporally extended relational database schemas used by the application programmers.” [70]

Bridging the two worlds: Perspective of this thesis

Before going further into process modelling, it is important that we digress a little from our ongoing discussion of prior work and highlight our perspective on several issues discussed so far. One of the fundamental shortcomings of prior work in our area has been to consider data modeling and business process modeling as two separate problems. However this thesis claims that when it comes to management of constraints within a database system, these two areas are intricately linked together. This assertion is central to this thesis and we revisit it in Chapter 3.

Observe that ER diagrams only tell us how data is logically structured in a database, which attributes to store for different entities, and some basic information about how these entities relate to each other. Similarly if we are to examine a business process model, we

will learn how data will be generated and what sequence of conditions need to be met before a business process concludes. However since the “status” of the process in question is itself inferrable from data contained in a database, there is automatically an implicit mapping between the data model and how a business process uses it. Furthermore the notion of time very elegantly connects the data model and the business process model in the context of constraints. Even though time is explicit in the data model (storing transaction and valid times) and implicit in the business process model (by observing that if a process “moves forward” then some time has elapsed), data itself is constrained by constraints possibly originating from both these sources (data model or the process model).

This subtle interplay between data modeling, temporality, and progression in business processes is why a critical examination of both data modeling and process modeling techniques is directly relevant to our work. Furthermore, this relationship gives significant credence to some form of temporal logic being an ideal tool for specifying process centric constraints over relational database systems.

BPMN and BPEL

BPMN (Business Process Model and Notation, see Figure 2.1 for an example) currently represents the state of the art in process modeling and has lately received significant attention from researchers and the business community alike. At its core BPMN is simply a process modeling notation very similar to flowcharts and UML activity diagrams. As of March 2011, the second iteratively improved version of BPMN is an official Object Management Group (OMG) standard [113]. Wohed et al., have exhaustively analyzed the capabilities of BPMN under the workflow patterns framework [172], and the book of White and Miers serves as a comprehensive reference guide for BPMN notation and usage [170]. Although BPMN may seem to be yet another process modeling language, there are several reasons why BPMN has revitalized the process modeling community. Foremost it addressed the criticism that UML activity diagrams (discussed shortly hereafter) have traditionally focused more on serving the needs of the software engineering community. The similarities between BPMN and traditional flowcharts gives this argument significant credibility. In fact, the case has been made that business process diagrams in BPMN are

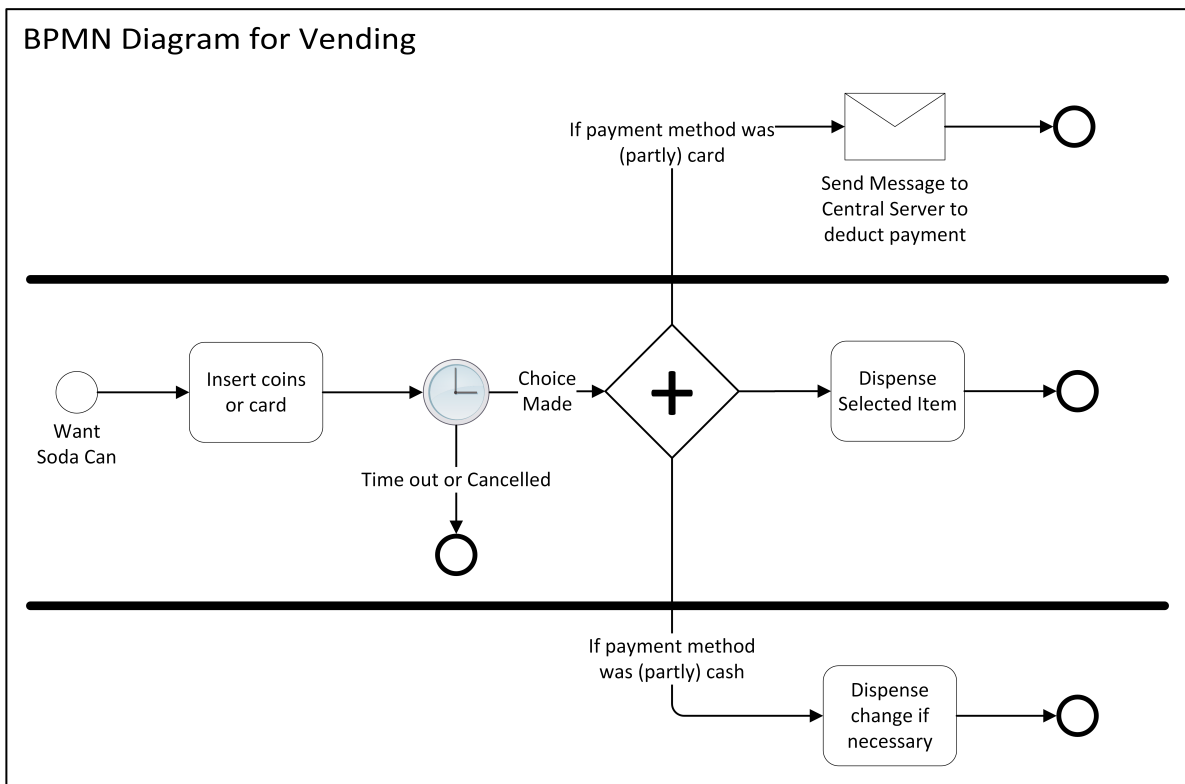


Figure 2.1: BPMN diagram for a vending machine. The diamond with a + sign represents a parallel split. See [112] for a detailed usage guide and many more examples.

much more amenable for human consumption [179]. Second, BPMN represents a unification of several notational languages that had been developed by various vendors to serve the needs of the process modeling community. The solidification of the BPMN specifications by the members of the OMG brought together many influential organizations and vendors of process modeling software to agree on a notational standard.

A parallel effort undertaken by the OASIS group has been to develop, standardize, and promote the Business Process Execution Language (BPEL) [71, 89]. BPEL can be best described as an XML based description of the control flow of a business process. The motivation for an XML based extensible language comes from the fact that many business processes today revolve around numerous independent web-services working together and passing information back and forth. Consequently a language for orchestrating (monitoring, managing, and enforcing) control flow of information between these services provides a platform for quickly building applications that support complex business processes. Van der Aalst et al. in their paper titled “Life after BPEL” [160], have made the criticism that on the surface BPEL looks like a simple messaging language that connects several web services (modules in the classical sense) together. However the authors do acknowledge there are specific features (albeit mostly syntactic) that make BPEL particularly appealing for mashing together web-services.

Because of the overwhelming support of the process modeling community that these two standards have received, there has been significant research conducted in mapping BPMN constructs into BPEL semantics. The objective of such research initiatives is to convert a business process diagram to its execution language specification and then perhaps to generate control flow constructs in various programming languages automatically so that a message passing/orchestration service can be easily built to monitor and enforce a business process. However these mappings are typically not complete and not sufficiently expressive to accommodate a wide range of control-flow situations. Furthermore, the mappings can often come with their own set of problems, including generating BPEL code that is either too complex and/or too unreadable by humans or that is not easy to trace back to the original control flow constructs. Nonetheless this is a research area that has the attention of the business process modeling community. The thesis of Jeroen Blox (2009) provides a comprehensive review of such techniques, their positive aspects, and shortcomings, and

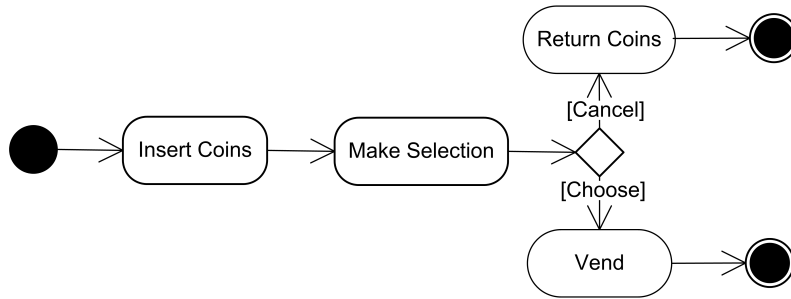


Figure 2.2: A simple UML activity diagram for a vending machine

it also proposes Blox’s own BPMN2BPEL translation mechanism [26]. It serves as an excellent starting point for those interested in research in this direction.

UML and OCL

Graphical modeling of data and processes has also seen significant interest from the software engineering community. 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 (OCL) [114], which allows simple invariants on objects to be specified. OCL can be considered to be a textual addendum to several types of UML diagrams. The motivation for OCL comes from the fact that objects, relationships, and activities specified within UML diagrams can be constrained, but neither is it feasible to forcibly integrate these constraints in one diagram nor can some of them be expressed visually. Consequently it is often better to express these constraints in plain text that accompanies a UML diagram. As an example let us say we have two types of objects in a specific UML diagram namely vehicles and owners (see Figure 2.3), and numerous data (attribute) specific constraints revolving around different types of licenses for different types of vehicles as well as different age requirements for different types of licenses. Instead of trying to express all this information, we can start listing object level invariants one by one, e.g., “if `vehicle.type = “Motorcycle”` then `IsTrue(LicenseClass(vehicle.owner.license), ≥, ‘M1’)`.” A brief explanation of what this constraint means (i.e., a vehicle’s owner must have the relevant license) can also be provided alongside. It is easy to see how a set of constraints

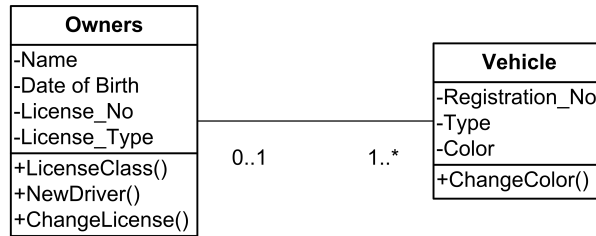


Figure 2.3: A simple UML class diagram. Note that the diagram is much richer than traditional ER diagrams and contains descriptions of functions. Other extensions can greatly enhance the usability of such modeling for software developers/engineers

written in text provides a much better way of conveying constraints than modeling them graphically.

Note that simple object level invariants, such as the one discussed above, can be easily converted into enforceable constraints in many target languages. With regard to SQL and database systems, Demuth et al. 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”¹ are available [51, 52]. Badaway and Richta [16] and Zimbrão et al. [178] have also presented similar mechanisms to translate OCL constraints to SQL assertions. Although such approaches to constraint generation may seem new in the context of the recently proposed OCL, in Section 2.3 we observe that template based constraint generation techniques for database systems were exhaustively studied in the mid 90s in the context of active database management systems. The very similar research conducted in the database community more than a decade earlier is perhaps not only more comprehensive and relevant, but also provides the foundations on which today’s object level constraints can be embedded in database systems.

A note on Object Relational (OR) Mapping

As alluded to previously, there are major differences between the way system developers working in the object oriented programming (OOP) paradigm view objects and the way database programmers view relational structures that store the details of those objects. In

¹more appropriately known as object-relational-mapping in the database community

practice the more we go beyond simple objects and tables, the more convoluted the mappings become, and the difference between these two world views become more and more irreconcilable. The view taken by database developers primarily focuses on efficient storage, management, and optimized retrieval/update of interrelated objects and the complex processes in which they partake. Furthermore relationships between objects are considered first class citizens in database design theory, complicating the matter further of how sub-objects of various classes may be related to each other. Database developers thus often disregard the notion of derived and sub-class oriented inheritance hierarchies and focus on normalization and separation of data based on actions and nouns. This in turn requires a complex web of wrappers to be written (and forever maintained) by a few programmers who are familiar with both the database schema and the object level view of data, so that objects and sub-objects can be properly maintained in the database by mapping their attributes properly at the storage level.

This mismatch has been well recognized as an irreconcilable difference by many prominent researchers in both the database and programming languages communities. The use of object-relational mapping tools to address these differences is a contentious issue as well. Having an object-relational mapping layer can often introduce its own set of complications: as a business and its database evolve, these mappings become more and more complex and time consuming to maintain. Some have even called this ongoing battle to bridge the two worlds as the “Vietnam of Computer Science” [108]. In this thesis we refrain from taking sides or even expressing an opinion on this matter. However we do point out that if an elegant and easy to use method of mapping complex objects and their behavior (processes) to storage level constructs in database systems is ever discovered, then a “policy-to-relational” mapping would be near trivial to support on top of that framework as the next logical step. For a deeper look at the problem of object to relational mapping and the general problem of arbitrary schema-to-schema mappings, the reader is directed to the works of Bernstein and Sergey [23], Adya et al. [5], Stajano [149] and O’Neil [115] as comprehensive surveys and excellent starting points for further research in this areas.

2.3 Logic, Constraints, and Active Database Systems

At this juncture we move from modeling of policies to enforcement of constraints in a database system. Note that there are several ancillary issues across the policy lifecycle that we have not yet addressed directly. Conflict detection and resolution are examples of problems that can either be addressed during the policy modeling phase or left for the execution engine as a run-time enforcement issue. More often than not, one of the fundamental objectives of having a logical policy model in the first place is to detect (or prevent) as many conflicts as possible before implementation. The role of the implementation engine in such cases is to cater to the exceptional situations where the model is not able to provide conclusive answers.

A more concrete example is that of a file on disk for which an operating system is managing permissions. A user and group based policy model may allow the group called “Administrators” access to an object and deny access to a particular user “John” for the same object. A trivial conflict could occur if John ever becomes part of the Administrators group, and consequently a logical policy model often comes with resolution mechanisms, such as giving preference to negative authorizations. We will examine several such issues throughout this thesis (conflict detection specifically in Chapter 5). In this section we present a broader examination of logics behind policy modelling, constraints, and method of enforcement in database systems.

Active Database Systems

An active database system is a DBMS with support for event detection (monitoring) combined with the facility to execute transactions autonomously [119]. Such a system is fundamental to the continuous monitoring of business rules and their enforcement. During the 1990s, active databases received significant attention from the database research community, and many prototypes such as Ariel [73], Starburst [171], SAMOS [68], HiPAC [47] and ODE [98] were used to demonstrate the applicability and practicality of triggers as a means for offering event-driven transaction processing. As a result of that research, most modern relational database systems today support the notion of triggers. A trigger

is a pre-programmed transaction that can be invoked when a particular condition in the database is met. User programmed triggers can also be executed based on temporal and periodic conditions as well as when certain failure conditions are met. Research related to trigger termination and confluence that emerged during the 90s forms the basis of reasoning over a set of event-condition-access (ECA) rules in a database. Since the solidification of the SQL3/SQL:1999 standard, triggers have become the de-facto means of supporting user programmed ECA rules within a database system [30].

Automatic generation of triggers to enforce rules in a database system is not a novel concept. Many of the fundamental constraints pertaining to data integrity, such as primary key, foreign key, uniqueness, and domain constraints, are, in fact, no different than triggers on top of relational tables [38]. Even if not explicitly employed by users, triggers are often transparently defined by a database system to monitor integrity constraints [31] and for the incremental maintenance of materialized views [72]. The use of triggers as a means to implement business level policies has also seen widespread adoption [30, 144, 95], and today most commercial database system have extensive support for user programmable triggers. In addition to supporting business rule enforcement, the most common uses of database triggers include alerting, replication (copying modified data to other locations), auditing (keeping a secure history of updates), and even logging access control events [30].

Although the research in this thesis does not directly concern itself with implementation of an active database system, it relies heavily on the fundamental framework for event-condition-action based rule checking provided by database systems. We believe that for a policy modeling framework to be useful, it must in some way have a direct (and preferably fully automatable) implementation path to be written as enforceable rules (triggers) within a database system. Furthermore we will rely on prior literature to propose optimizations with business policy, discover conflicts, and propagate knowledge about policies (as manifested as triggers/rules) back up to the business level modeling layer.

Since the proposed policy modeling language of choice in this thesis is first order temporal logic of the past and its metric extensions, the work of Chomicki [33] and Toman and Chomicki [154, 34] is of significant importance, as it lays down the foundations of efficiently implementing first order linear temporal integrity constraints within relational database systems. It is worth acknowledging that the aforementioned works still remain state of the

art in describing the limits of the types of database level constraints that can be efficiently enforced and the compromise between auxiliary space and efficiency that needs to be made during implementation. The performance results presented in this thesis (Chapter 4) serve as yet another practical validation of the above works.

Temporal Logic and Policy Models

Research pertaining to the use of temporal logics in computer science has spanned a period of over five decades. However among the various fragments, the linear propositional fragment of temporal logic (LTL) has received significantly more attention than others. The popularity of LTL is attributable not only to the fact that computer aided verification and model checking have their theoretical underpinnings in LTL, but also perhaps to the simplicity and elegance of LTL. The decidability of propositional LTL was first shown to be P-SPACE complete by Sistla and Clarke [147]. Recent results have identified several broader classes of first order metric monodic variants of temporal logic as being decidable [118, 79]. This thesis directly uses these results to make claims about several problems, such as conflict detection within a constraint system. A reader who appreciates topics such as satisfiability and decidability of logics will consequently appreciate the reduction of several problems in constraint management to these well studied problems. A large body of work in the area of policy specification using temporal logic, including this thesis, is motivated by the classical work of Lamport (Temporal Logic of Actions [97]) and by the seminal work of Pnueli [123, 122].

Using LTL as a policy level tool for specifying temporal access control restrictions [88] and determining regulatory compliance using historical audit logs [57] have received varying degrees of attention from researchers. More recently, there has been a push to use metric first order temporal logic as a unifying layer of policy that is able to support obligation modeling, Chinese Wall policies, and even data retention policies [19]. However, to the best of our knowledge, no prior approaches have looked at the problem of integrating data models and process models to enforce process centric temporal integrity constraints within database systems. As we shall see in subsequent chapters, there are many challenges faced when porting a logical policy model into its implementation, which prior works have failed

to address.

Several researchers have examined the stark resemblance between the problems of model checking, satisfiability, and variants of the constraint satisfaction problem (CSP) [24, 169, 62]. The types of constraints examined in the work of Demri and D’Souza [50] are very similar to those that we examine in our work and that we expect to encounter in business situations. Their primary result shows that solving constraints specified in linear temporal logic over certain decidable SMT theories (more specifically the constraint systems $(\mathbb{N}, <, =)$ and $(\mathbb{Z}, <, =)$) is equivalent in computational complexity to that of classic LTL satisfiability. The link between several problems in database systems and the constraint satisfaction problem is also well known to researchers. Many classical problems, such as containment for conjunctive queries and query answering using views, can be directly reduced to that of constraint satisfaction [166, 94]. Once again, we will rely on these classical results to answer several important questions about what class of process-centric constraints can be accommodated in our proposed model.

If we step further back, we can see that there is also a significant link between business process modeling techniques, such as workflows, and state machine oriented analytical techniques such as Markov Modeling and Petri-Nets. Pesic and van der Aalst have examined this very resemblance between finite state machines and business process models [121]. However, because of its sheer complexity, these automata-theoretic models have had little or no impact on the process modeling community. With that being said, analysis of state oriented systems does relate directly to several problems in policy verification and must be considered when proposing and analyzing policy models.

Finally, in the context of our own work, we note that the notion of modeling a business situation as a state machine and then expressing (and checking) path constraints is very closely associated with formal verification and model checking [36, 22]. There is, in fact, a direct correspondence between traditional model checking of software systems (specifying invariants in linear temporal logic, checking for liveness and safety, etc.) and our proposed model for ensuring business process integrity. In the context of formal verification, the objective is to verify properties of a system over all possible runs. Although in our work the lines between a model and its properties (constraints) are blurred, there is a significant overlap in the context of identifying potential conflicts in the business lifecycle where a

constraint may be violated.

Chapter 3

Policy Modelling for Relational Objects

3.1 Outline

In this chapter we propose a novel way of bringing together business processes and database constraints using first order temporal logic. In Section 3.2 we discuss the assumptions we are making about the business situations and databases that must hold for our policy modeling framework to function. In Sections 3.3 and 3.4 we present a formal description of our policy model and relate it to integrity constraints specified in linear temporal logic (LTL) over the history of business objects contained in a database.

Section 3.5 introduces our proposed state-oriented, graphical policy/process modeling technique for relational database systems (which we call the database constraint modelling language, or DCML). It is shown that the language is able to represent constraints specified in the past-only fragment of linear first order temporal logic. Section 3.6 discusses several novel aspects of the language and demonstrates, using micro-examples, that a broad class of process-oriented description of business processes can be easily translated into DCML models. In Section 3.7 we revisit prior research that is most closely related to the language presented in this chapter and contrast its features with those of prior approaches. Finally,



Figure 3.1: A database is a reflection of the complete picture of a business

Section 3.8 concludes this chapter by summarizing its contents and discussing the next logical step of converting constraint models into actual implementable constraints.

3.2 Premise

Before describing our underlying model for policy enforcement in a relational database, it is important to draw out a distinction between business policies at the conceptual level and their equivalents at the database level (Figure 3.1). In most cases the contents of a database represent the record keeping aspect of compliance with a given set of policies. To explain this further, let us consider an example where “client consent” is required by a business for a particular situation, and it is recorded in a database as a binary field (yes or no). Recording the fact that consent has been received in a database usually serves as an audit or perhaps an internal check for business applications/processes that require the client’s consent. If the client requests a service for which his/her private information needs to be disclosed to third parties, automated systems can check whether the client’s consent to disclose personal information has been received or not, by simply querying the database. Thus, the contents of the database reflect the ground truth on which policy decisions are taken.

The duality between what is recorded in a database and the current state of a business is critical in our framework. We assume that both the database and a business are, in a manner of speaking, equivalent state machines that move in lock-step with each other. In our proposed framework, the following two assertions must hold true in all situations

where policy-centric mappings to a database rule system need to be established.

Assertion 1: *A database state corresponds to a business state and there must always exist a describable relationship between a change in the business state and the associated change in the database state.*

We define a database state as the contents of a database containing historical and current operational records of a business at a specific point in time. Similarly, we define a business state as an abstract union of the states of all business objects, processes, and situations that are ongoing and encountered in the past. We use the notion of “correspondence” informally to assert that the changes to a database and changes at the business level (for example progression in a particular business process) must always be related to each other. For example, if the contents of a database are impacted by a transaction, then we must have (in some way) changed the state/reality of the business as it exists. Similarly, if the business state has changed, then that change is immediately made persistent by updating the contents of the database. In other words, the state of a business and the state of a database not only relate to each other but also change in sync with each other.

To further formalize the duality, we require that there must exist a correspondence (“describable relationship”) or a mapping between state changes at the business level and state changes at the database level. This correspondence may be informal and does not need to be explicitly documented in natural language, as long as it can be provided by a business level user or a database administrator. Intuitively this requirement is intended to ensure that there must exist a sound explanation or a systematic method of explaining business level concepts and how those concepts behave over time as stored in a database. This is a critical requirement in actualizing a business policy within a database and conversely storing and monitoring associated rules within a database system. In our example of client consent, the relationship may be described by a database administrator or a corporate privacy expert by interpreting a binary attribute (0 for no consent received and 1 for consent given). It can perhaps further be elaborated by those experts that changes to consent (business level state changes of giving or taking away consent) are equivalent to changes in this binary value associated with each client in the database.

Assertion 2: *A policy rule is a path restriction specified in the business state space and*

corresponds to a database constraint which is a path restriction specified on the database state space.

Once we have established that state changes at both levels (the business concept and database levels) are essentially the same and a reasonable explanation is available for any and all such changes, a policy can then be described as a sequence of required (or instead undesirable) state changes in the business level. For example, a policy could require that a business state in which a customer’s information is disclosed to a third party without prior consent should never be reached. Since business rules can be described as legal (or illegal) state changes at the business concept level, we can now work towards translating these rules as legal (or illegal) database state changes.

We believe that there are two main advantages of a state-based mapping between logical policy models and relational database systems. First, we note that databases are by design complex state machines and typical policy-centric decision making (access control, integrity checking, etc.) occurs when transactions attempt to access the database in a particular state or move the database from one state to another. This reasoning applies to the business world as well, since most traditional business processes progress towards their conclusion in discrete steps. Second, reasoning over state transitions is a well understood problem in the context of model checking. Formal properties of state system models, for example, safety (something bad should never happen) and liveness (something good should eventually happen) can be easily specified in linear temporal logic and tested against a formal model of the policy. As we explain in Chapter 4, these temporal properties can also be automatically translated into database constraints, thereby making implementation of the originally modeled business policies a seamless process. Thus, this state system formalism is ideal to capture (make formal) the semantics of a business process.

3.3 Enforcement Model: Moving from Process Models to Constraint Models

So far, we have used the term “business state” in a very abstract sense to mean the entire business, and the term “database state” has been used to describe the contents of the

entire database. However, as pointed out in the introduction of this thesis, we rely on individual, pre-existing, visual, state-oriented descriptions of business processes (such as workflows, process models, state charts, etc.) to map out fragments of the business state space onto fragments of the database state space and then constrain the database state space appropriately. In other words, instead of attempting to relate the entire business to the entire database, we restrict ourselves to specific well-understood functions of a business (and the governing policies associated with those business functions).

The reason for doing so is straightforward. A typical business user can understand the broad-ranging assertion that a “business is equal to its database.” However, in practice, the possibly hundreds and thousands of correspondences between various business processes and individual tables and rows is likely to be beyond the scope of a single user to comprehend. In a typical commercial database deployment, no single user can describe completely and accurately all the complexities in all activities of a business. Similarly, it is equally likely that a single DBA may not be aware of (or be able to explain) the behaviour of all parts of a database as it changes over time. Therefore, instead of attempting to map out two very large and abstract state spaces and then translate rules from one to the other, we reduce these state spaces so that they are more manageable.

Reducing the state space to smaller and easier to understand fragments has several advantages. Foremost, visual descriptions of processes are typically readily available, as they are widely used tools in the businesses and software engineering communities. And even if processes are not fully documented, business users can provide pictorial descriptions of business processes (stages, progression through the use of arrows, branches for various exceptions, termination conditions, etc.) at a very high level, which can then be used as a basis for database constraints. Thus, such descriptions play the role of providing the “business state space” for individual business processes in our model. Furthermore, these process diagrams resemble state machines (see Figure 3.2) and provide significant insight into the valid (required) behaviour of the associated parts of the database. Finally, we note that at the level of individual process/workflow/activity, these descriptions are easy to understand. At the individual process level, the correspondence between changes in the business level objects and database constructs is much easier to capture. Consequently the overall process of translating business rules into database constraints is made much easier

when using one process diagram at a time to guide implementation.

3.3.1 Terminology and Setup

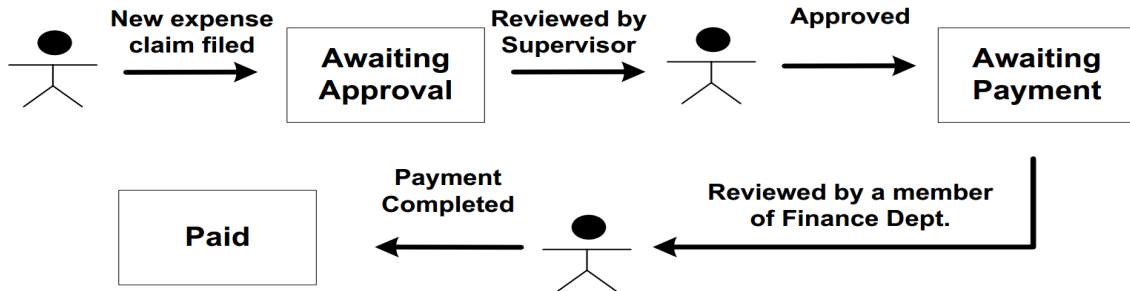


Figure 3.2: A hypothetical process model for expense claim processing describing the stages (states), actors (users) and actions (transactions) related to an expense claim.

Our overall goal is that of translating constraints from the business world to the database world. Our approach, as described above, is to do this one process (business function) at a time. To make our notation consistent, from this point onwards we will use the term *artifact type* or (*business*) *object type* to refer to a concept about which various processes are specified. For example, Figure 3.2 is a simplistic example of a “payment process” on the “expense claim” artifact type. Such process diagrams often embed within them interpretations of the business policy for the process being outlined. For example, a policy rule embedded in the above diagram could be that “only a member of the finance department can issue a payment for an expense claim that is awaiting payment.” Consequently, all individual artifacts of that type (all expense claims) must follow the rules embedded in the depicted process. Note that we are using the term *artifact* as it has been used in prior work¹ to mean, broadly, an object of interest to a policy maker.

Similarly, we are using the term *process model* to represent the large category of visual state-oriented descriptions of operational business specifications, as discussed in Chapter 2. Although these specifications can include a broad range of diagrams, it is best that

¹This is examined in greater detail in Section 3.7. The reader unfamiliar with this term can consider an artifact to be a holistic view of data relevant to a business activity or process.

the reader consider these descriptions to be informally specified workflows or flowcharts (such as the example in Figure 3.2). A more rigorous examination of issues and features of process description languages that make them more amenable to conversion into database constraints is presented in Chapter 7. However, for now, the reader should consider them to be informal state oriented diagrams that have to be given formal meaning as database constraints. Therefore, we use the term “policy makers” to refer to users who will convert these informally specified business into database constraints.²

Finally, recall that having these process diagrams is not a strict requirement for constraining a database system, as long as a detailed explanation of the behaviour of the business processes is available. It is of course helpful if this description is available in visual notation (state oriented and depicting valid progression), as this makes the task of converting process models into database constraint models significantly easier. Clearly, if the business states can be easily identified in the database (e.g., the notion of being in the unpaid business level state maps onto a binary paid flag being zero in the database), then we can begin to reason about paths an artifact can (should) take across the business state and how they map onto restrictions on the database state space. However, we point out that a proficient enough policy maker (who is knowledgeable about a process on an artifact type) may not need to rely on business level process diagrams to embed constraints on a database.

3.4 Artifacts in Relational Databases

The first step in converting business rules into database constraints is that a user must identify artifacts within a database system. The most natural way to do this is to use the abstraction provided by database views.

Definition 1. *An artifact type is a user-defined view over a database schema that captures all the data required to enforce a business policy.*

Definition 2. *An artifact is a uniquely identifiable row in an artifact type.*

²as opposed to a corporate level policy developers that interpret legislation to develop high level policy statements

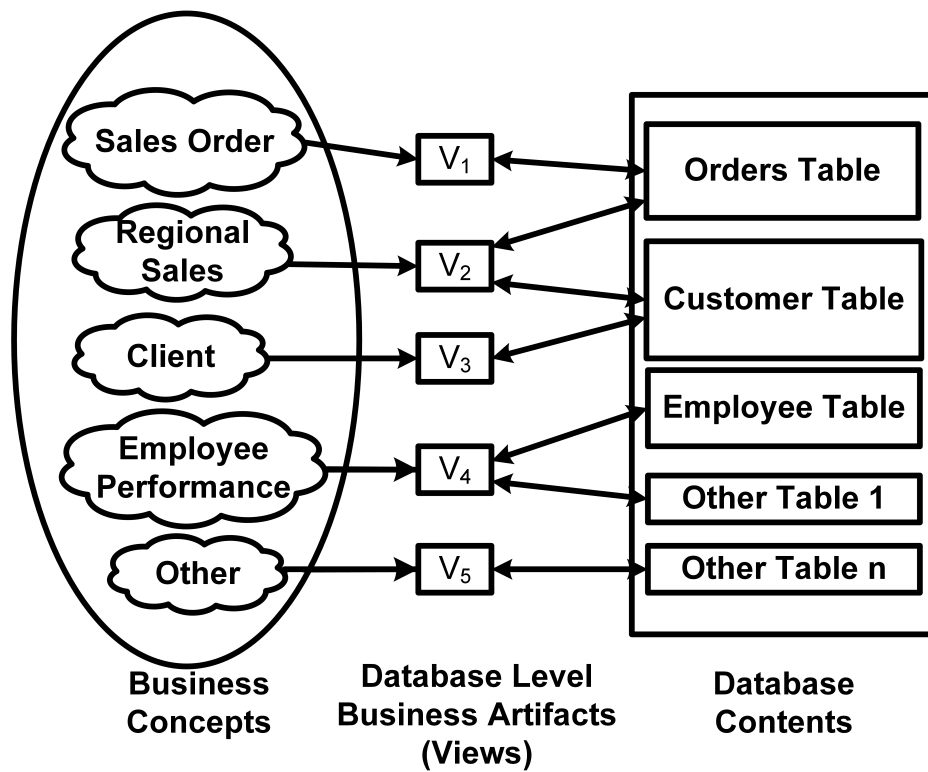


Figure 3.3: Artifact types (views) essentially serve as a mapping between business concepts and the physical contents of a database system

In relational database systems, views are a flexible concept that are able to encapsulate an arbitrary business object or any collection of individual pieces of information that relate to a higher level business function. Consequently, a view will define an artifact type, and all individual artifacts will therefore be elements in the view. To elaborate this concept, let us consider an e-commerce database that stores transactional information regarding customer sales and purchases. The Orders, Customers, and Shipments tables in such a database are assumed as follows:

Customers: (int CST_ID, varchar NAME, varchar ADDRESS, ..., varchar ADDRESS_COUNTRY, varchar TYPE)
Orders: (int ORD_ID, int CST_ID, int SHIP_REF, datetime DATE_TIME, float AMOUNT, bool PAID, bool APPROVED, ..., varchar NOTES)
Shipments: (int SHIP_REF, bool SHIPPED, varchar SHIP_METHOD, ..., datetime SHIP_DATE)

The intent of the Orders table is to store the details of all sales that the business handles. Each row in this table is uniquely identified by its ORD_ID attribute, and the remaining attributes represent detailed information about the particular order. The Customers table simply stores the name and address for customers in the database that can place one or more orders, and the Shipment table is intended to store the logistical information associated with the orders and customers in the database system.

Within the above schema the user may define several artifact types as views. For example, the views defined by the following SQL queries are all valid definitions for an artifact type SalesOrder:

(a) SELECT * FROM Orders NATURAL JOIN Customers NATURAL JOIN Shipments
(b) SELECT * FROM Orders NATURAL JOIN Customers WHERE AMOUNT >= 100 AND AMOUNT <= 1000
(c) SELECT * FROM Orders NATURAL JOIN Customers NATURAL JOIN Shipments WHERE ADDRESS_COUNTRY = "CANADA" AND SHIP_METHOD = "UPS"

Note that from a business perspective each row in the above views may be uniquely identified as a sales order artifact that was (or is currently) being processed by the business.

Views (b) and (c) impose criteria on the artifact types being defined and therefore will only contain SalesOrder artifacts that match their respective predicates. Although all of the above are valid definitions of business artifacts, there may be scenarios where each of these definitions may serve a special purpose for an individual policy maker. For example, view (a) may be used to enact policies that apply to all sales orders, but view (c) may be of specific interest to a regional sales manager for Canada, who may want to impose specific policies on orders that are shipped to Canada via UPS. Also note that since artifact types are view definitions over the schema, individual artifacts are not necessarily physically stored in a database. Consider a more complex artifact type, SalesTotals, defined by the following view generating query:

```

SELECT
YEAR(DATE.TIME), ADDRESS.COUNTRY, SUM(AMOUNT) TOTAL
FROM Orders NATURAL JOIN Customers
WHERE PAID = true
GROUP BY YEAR(DATE.TIME), ADDRESS.COUNTRY

```

Observe that each artifact in the above view will represent the yearly sales for a particular country that may be derived from many individual rows that contribute towards the total. Such artifacts may be relevant to marketing and advertising policies that target customers in regions where sales are low.

3.4.1 State of an Artifact

For each object/artifact type contained in a database, policy makers (or their delegates) may define arbitrarily many boolean functions that group individual artifacts together for policy purposes. This implicitly defines a state space in which states correspond to subsets of the functions that evaluate to true (when applied to individual artifacts). However, rather than considering all functions simultaneously, we propose the following definition:

Definition 3. *For any boolean function F defined over the attributes of the artifact type, an artifact is in state F , or is in the F state, if F evaluates to true.*

Note that an individual artifact can be in many different states at the same time, provided it satisfies the functions associated with each of the states. For example, consider

the view definition (a) from the example presented earlier. A function on the attributes of this view could be defined as follows:

```
DEFINE STATE Unpaid (SalesOrder)
WHERE (PAID = false AND REVIEWED = false AND SHIPPED = false)
```

Implicitly, the meaning of this functional state definition is that a sales order artifact is in the Unpaid state if all three of its paid, reviewed, and shipped attributes are false. Similarly, we can define states Paid and Shipped as follows, where an artifact may be in neither, one, or both of these states:

```
DEFINE STATE Paid (SalesOrder)
WHERE (PAID = true AND REVIEWED = true)
```

```
DEFINE STATE Shipped (SalesOrder)
WHERE (shipped = true)
```

States for more complex and derived artifact types can be defined in precisely the same way. The following defines the state LowSalesVolume for the SalesTotal artifact type as one where the total sales volume is less than 10,000:

```
DEFINE STATE LowSalesVolume(SalesTotal)
WHERE (TOTAL <= 10000)
```

We can summarize the notions of artifact types and states as follows. Views are used to define artifact types or object types. A single artifact or object is an element in the view. A named boolean function defined on the attributes of the artifact type is used to determine whether a particular artifact belongs to that named state, and the states of an artifacts are dependent on the particular assignment of its attributes. For example, if x is an artifact of type V , or equivalently $x = (a_1, a_2, \dots, a_n) \in V$, where V is a view over a database schema, then for a labeled state S with its appropriate conditional function, x is said to belong in state S if and only if $S(a_1, a_2, \dots, a_n)$ evaluates to *true*. To further simplify our notation, we will write $x \in S$ if and only if $S(x)$.

3.4.2 Restricting the Database State Space

Once policy makers have identified business artifact types and further defined states of interest that individual artifacts can be in, all that is required is to establish the associated restrictions in the state space. Note that under this setup we have reduced a potentially very large database state space to a compact, user-defined state space specific to a business artifact type being examined. Moreover, since this state space at the database level was quite possibly derived from a workflow or process model, it should be similar in size (number of stages in the workflow) and behaviour (legal/illegal paths in the business level workflow). The only task left is that of establishing path restrictions within the database artifact state space that mimic policies and legal paths in the business process model.

To accomplish this we rely on classical first order temporal logic [122]. An example of a temporal constraint on the Sales Order artifact type defined earlier could be that “an artifact can only reach the *paid* state if it was previously in the *approved* state,” where both *paid* and *approved* are boolean functions defined on the artifact type. The motivation for restricting state transitions for specific artifact types comes from the fact that these restrictions are embedded in process models such as the one presented in Figure 3.2 on page 50, and it is these flow oriented constraints when fully captured that delineate a business process.

3.4.3 Formal Model

Using terminology that is consistent with prior work [122, 97, 19, 123, 24, 79, 147], we provide a formal interpretation of our policy model that employs temporal integrity constraints over uniquely identifiable tuples in database views. Since we are considering constraint enforcement over a temporal database (and temporal objects therein), we assume the presence of the infrastructure to track changes to the attributes of individual artifacts.

In our model we consider the domain of time to be isomorphic to a finite set of ordered natural numbers $(0, 1, 2, \dots, c)$, where c represents the current or most recent transaction time. Each artifact has its own finite history that can be considered as a temporally ordered sequence of changes made to the attributes of the artifact since its inception:

Definition 4 (Artifact History). *The history of an artifact is a mapping from natural numbers representing time to ordered tuples of the form $\{(t, a_{1t}, a_{2t}, \dots, a_{nt})\}$, where each a_{it} is the value of the attribute a_i at time t .*

Therefore, the history of an individual artifact with n attributes can be considered an ordered sequence of tuples starting at $t = 0$ (artifact inception) and ending at $t = c$ (current or most recent time). Note that artifact histories are not combined or intertwined in anyway. That is, each individual artifact history (timeline) is retained independent of other histories in the database. We use the shorthand notation of depicting the history of an artifact A as: $A_0, A_1, A_2, \dots, A_c$. The same can be viewed in its expanded form as follows:

$$\begin{aligned}
 A_0 &: (0, a_{10}, a_{20}, \dots, a_{n0}) \\
 A_1 &: (1, a_{11}, a_{21}, \dots, a_{n1}) \\
 A_2 &: (2, a_{12}, a_{22}, \dots, a_{n2}) \\
 &\dots \\
 A_c &: (c, a_{1c}, a_{2c}, \dots, a_{nc})
 \end{aligned} \tag{3.1}$$

Approaches to maintain individual artifact histories (e.g., auxiliary tables and delta-encoding) are examined in detail in Chapter 4. However, for now we assume that a database is capable of tracking changes to individual artifacts and utilize the above abstraction for constraint enforcement.

As is the case in implemented database systems, the number of attributes of the artifact type, n , is finite and the domains of each of the a_i are assumed to be finite as well. Thus, there is always a finite amount of history that is contained in a database for every artifact.

Definition 5 (Constraint). *A constraint in our model is a Linear Temporal Logic (LTL) formula defined using the attributes of an artifact type but interpreted (enforced) over the history of all artifacts of that type.*

All artifact histories must comply with all the constraints defined on their respective artifact types and a constraint on an artifact type cannot be selectively enforced on a few artifact histories only. To specify constraints we use LTL formulas with past connectives

only (Past-LTL). This is a natural choice in database systems, since enforcement decisions can be made with certainty by looking at the past history of an artifact only. The interpretation of these constraints (i.e., the enforcement model discussed in detail in Chapter 4) can be summarized as follows: Given an update to an artifact A , we consider the history entry that *would be committed to disk* if this update were allowed to happen. That is, we check for each artifact being impacted that its anticipated history $A_0, A_1, \dots, A_{current}, A_{next}$ is a valid trace for all the constraints specified over A 's artifact type. If the resulting trace is valid, then the transaction is allowed, i.e., A_{next} is accepted as the new history entry for the artifact, otherwise the artifact stays unchanged as $A_{current}$.

To accommodate restrictions over different domains, constraints are specified using *Constraint Linear Temporal Logic* (CLTL) [173, 17, 24, 49, 48, 50]. CLTL extends LTL and adds a spatio-temporal aspect to the logic by allowing constraints to be specified over domains with well-defined operators as propositions in the logic. For example $\bullet(x < y)$ is a CLTL formula interpreted over the domains of x and y and a definition of the operator $<$. We use traditional operators ($<, \leq, =, \neq, \geq, >$) for well known (finite fragments of) domains $\{\mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathbb{Q}\}$ to specify constraints. Operators on complex domains, such as ($=$) on the domain of Strings (varchars in SQL), are used in their intuitive programmatic interpretation (i.e., string comparison). Note that we can adopt this interpretation because our domains are assumed to be finite. Consequently, we can treat database level attribute types (e.g., varchar/strings, floating point) as finite subsets of a well ordered domain, such as \mathbb{Z} , and then define operators such as equality to mimic the behaviour of the appropriate database functions.

The definitions of the past LTL operators utilized to constrain the history of artifacts are consistent with prior literature and are as follows:

1. CLTL formulas $S_A, \neg S_A, S_S \wedge S_B$ and $S_A \vee S_B$ evaluate as true iff the conditions, specified as constraints over the appropriate domains, hold true
2. $\bullet S_A$ (previously S_A) is true at instant t iff $t > 0$ and S_A is true at instant $t - 1$
3. S_A **since** S_B is true at instant t iff for some $s, 0 \leq s < t$, S_B is true at instant s , and for every $u, s < u \leq t$, A is true at instant u .

4. $\blacklozenge S_A$ (sometime in the past) is true at instant t iff *true* **since** S_A at instant t
5. $\blacksquare S_A$ (always in the past) is true at instant t iff $\neg\blacklozenge\neg S_A$ at instant t

Thus, constraints specified using CLTL are extended LTL constructs and are interpreted in the same manner as classical LTL constraints. It is important to recognize that constraints are specified over an artifact type but enforced on all individual artifacts' histories of that type. For example, consider the following constraint defined on the sales order artifact type defined earlier:

$$paid \Rightarrow \bullet(\neg paid \wedge amount > 0)$$

It forms a temporal constraint using two attributes from different domains (boolean and floating point, as derived from the view specifications). It specifies that if an artifact's paid attribute is currently true, then it must be the case that previously the same boolean attribute was false and the amount attribute was greater than zero. Note that this constraint is specified using the attributes of the artifact type and then enforced on all artifact histories.

It is important to recognize that the CLTL constraints are always interpreted over the history that would result as a consequence of the "next" entry being added. That is, a given constraint is always checked over $(A_0, A_1, \dots, A_{current}, A_{next})$. In the above example, if a new entry in the artifact history is being appended (A_{next}) in which the boolean attribute paid is true, then condition $\neg paid \wedge amount > 0$ must hold on $A_{current}$ before A_{next} is allowed into the history. In other words, enforcement is done by continuously monitoring the impact of the next (to-be-written) entry in the artifact history and ensuring that the resulting history is compliant with the constraint specified.

As is common in database systems, we adopt a strict model of compliance, where all the user-specified CLTL constraints on all artifact types must independently hold for all individual artifacts. This is especially relevant to situations where a database transaction modifies several different artifacts at the same time. In such cases we require that each of the individual histories being appended remains compliant with all the constraints defined over their respective artifact types.

3.4.4 Starting and Ending Artifact Lifecycles

Time in our model is finite and isomorphic to the natural numbers (including $t=0$), and the temporal operators in LTL refer to time implicitly. Within this model we require a mechanism to constrain the way artifacts can be created to start their lifecycles. More formally, we need a mechanism outside traditional LTL to restrict the values of the artifact explicitly at $t=0$, the time at which the artifact is created. To do so, we define a special *non-temporal constraint* $\phi_S(A)$ that is interpreted for the artifact at $t=0$ only. Users can define $\phi_S(A)$ as the function that represents the starting state of an object. For example, consider the following definition of $\phi_S(A)$:

$$\phi_S(\text{SalesOrder}) : \neg\text{reviewed} \wedge \neg\text{paid} \wedge \neg\text{shipped} \wedge \text{amount} > 0$$

Note that since $\phi_S(A)$ is required to be non-temporal, it cannot (and in the above example, does not) contain any temporal operators. This is because the definition of $\phi_S(A)$ is a constraint only on the artifact at time zero (A_0) and can be interpreted as an initialization requirement for all artifacts of a given type. In the above example, the initialization requires that all Sales Order artifacts when logically created (at $t=0$) must not be reviewed, not be paid, not be shipped, and must have a non-negative amount. Also note that defining $\phi_S(A)$ is optional, as policy makers may not want to restrict the way artifacts begin their lifecycles. Consequently, in such cases $\phi_S(A)$ is defined as the *true* function, and therefore all artifact histories trivially comply with $\phi_S(A)$ allowing artifact attributes to take any values at $t=0$.

Conversely, policy makers may want to consider the logical deletion of artifacts and therefore constrain that special time in the history in some way. To denote the “end of an artifact’s lifecycle,” we transparently augment the definition of an artifact to include a termination attribute/marker denoted as ϕ . The history of the artifact is thus now defined as the following sequence:

$$\{(t, a_{1t}, a_{2t}, \dots, a_{nt}, \phi)\}$$

where each a_{it} still denotes the value of the attribute a_i at time t , but a new boolean attribute ϕ denotes whether the history of the artifact, at t , has been terminated or not.

At any history entry, if ϕ is false, then the object has (at that point in time) not been logically erased from the database, and therefore its history can be appended further. If, however, ϕ is true, then the object is considered to have concluded its lifecycle and “actions” or “changes” to it (i.e., an A_{next} different from $A_{current}$) should be rejected. In other words, once an object reaches its “dead” or termination state, signified by ϕ being set to true, it should forever be preserved in that “dead” state. To accomplish, this we transparently add two CLTL constraints for every artifact:

$$(1) : \bullet\phi \Rightarrow \phi$$

$$(2) : \phi \Rightarrow \bigwedge_{a_i \in A} \bullet a_i = a_i$$

The net result of these two constraints is that (1) if previously the ϕ marker was set, then it must remain set and (2) all attributes retain their previous values when ϕ is set to true. As a consequence of these constraints, we eliminate the possibility of changes to the attributes of the artifact from the moment it logically ended its lifecycle. Note carefully that constraint (2) is also applicable at the first time instance ϕ is set true, and thus it ensures that if an artifact’s lifecycle is being terminated, then that is the only event happening. In other words, constraint (2) implies that an artifact can not change attribute values and set its lifecycle termination marker at the same time.

Also note that since ϕ is introduced transparently, it is not directly available to policy makers and therefore cannot be used to define additional constraints. In subsequent sections we will provide an interface through which rules about lifecycle termination can be implemented by allowing the use of ϕ indirectly and in limited circumstances. The mechanism we have adopted (end of lifecycle marker) is similar to prior work where history during the “logical existence” of an artifact is forever preserved. Other approaches, such as deleting the entire history of the object at lifecycle termination, are also possible (i.e., we could remove all traces of the artifact ever existing in the first place). However, we do not explore such alternatives in this thesis.

3.5 State Transitions and Database Level Workflows

In this section we introduce a visual constraint modeling language for artifact lifecycles represented in a database. Note that in an unrestricted database, one with no rules or policies, individual artifacts can change freely, and therefore their attributes and membership in each of the pre-defined states can change freely from one point in time to another. We repeat once again that the purpose of using a mechanism such as CLTL is essentially to specify a particular sequence of artifact attribute value changes as being valid or invalid, which in turn will reflect how individual artifacts can change state. Our proposed visual notation mimics statecharts and sequential workflows but also establishes a one-to-one link between restrictions in first order logic and restrictions in the visual notation for database centric artifact lifecycle management.

3.5.1 Visual Constraint Notation

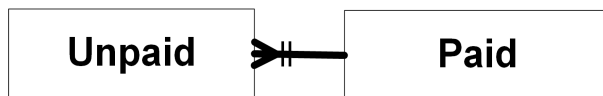


Figure 3.4: Simple Exit Restriction between the Paid and Unpaid States

Our diagrammatic convention for specifying database level CLTL constraints resembles traditional workflow and process modelling techniques in that the notion of a ‘state’ resembles that of a stage in a business process. One such state is represented by a labelled box or rectangle on a constraint diagram. As described earlier, an artifact (representing an individual instance of a business process) is in a given state if it satisfies the function defined on the attributes of the artifact type to be in that state. We use lines with varying arrows (connectors at different ends) as connections between the states to imply some path restrictions associated with two states. For all such connections there is formal interpretation as a CLTL constraint on the artifact history.

Consider the diagram in Figure 3.4 involving two states on SalesOrders that we had defined earlier. For brevity we have omitted the actual function definitions associated with

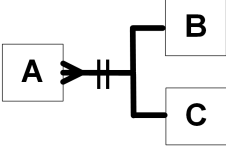
the states and instead referred to them via their state name/label. There is a connector with a three pronged arrow at the Unpaid side and a direct connection to the Paid state. The logical interpretation derived by this arrow in the diagram can be read from Table 3.1: If a SalesOrder artifact is in the Unpaid state at time t , and does not satisfy the condition to be in the Unpaid state at time $t + 1$, then it must satisfy the condition to be in the Paid state at time $t + 1$. The LTL equivalent can also be specified by simply reading Table 3.1 as follows:

$$(\bullet \text{Unpaid}(\text{SalesOrder}) \wedge \neg \text{Unpaid}(\text{SalesOrder})) \Rightarrow \text{Paid}(\text{SalesOrder})$$

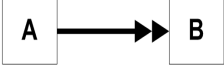


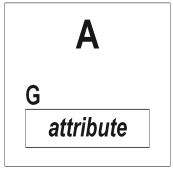
If we unwrap the states using their definitions, we can write the temporal constraint as the following implication on the boolean attributes of the view:

$$(\bullet (\neg \text{paid} \wedge \neg \text{reviewed} \wedge \neg \text{shipped}) \wedge \neg (\neg \text{paid} \wedge \neg \text{reviewed} \wedge \neg \text{shipped})) \\ \Rightarrow (\text{paid} \wedge \text{reviewed})$$

The notation presented below represents the basic constructs of our proposed graphical database constraint modeling language (DCML):

Visual Notation and LTL	Interpretation
<div style="text-align: center;">  </div> <p data-bbox="240 1388 618 1419">Multi-Way Exit Restriction</p> $ \begin{aligned} &\bullet A(x) \wedge \neg A(x) \\ &\quad \Rightarrow \\ &B(x) \vee C(x) \end{aligned} $	<p data-bbox="651 1209 1406 1652">If an artifact 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 or state C at time $t + 1$. Note that the graphical notation only depicts a 2-way exit restriction leading to states B or C, but the notation can be extended and used to specify exit restrictions in which the OR conditional arrows end towards n different states where $n \geq 1$. In such cases the artifact must satisfy the condition to be in at least one state on the right hand side at time $t + 1$.</p>

Continued on next page

Visual Notation and LTL	Interpretation
 <p data-bbox="310 548 548 579">Entry Restriction</p> <p data-bbox="237 590 613 625">$(B(x) \wedge \neg \bullet B(x)) \Rightarrow \bullet A(x)$</p>	<p data-bbox="651 386 1406 693">If an artifact 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 $t - 1$. Note that this constraint can be considered the dual of the exit restriction. A multi-way entry restriction can be created by specifying several entry restrictions originating from one state and ending in many others.</p>
 <p data-bbox="264 846 594 877">Never Eventually Reach</p> <p data-bbox="318 888 540 924">$\blacklozenge A(x) \Rightarrow \neg B(x)$</p>	<p data-bbox="651 810 1406 888">If an artifact was ever in state A, it should never satisfy the condition to be in state B.</p>
 <p data-bbox="337 1136 526 1167">Disallow Exit</p> <p data-bbox="321 1178 526 1213">$\blacklozenge A(x) \Rightarrow A(x)$</p>	<p data-bbox="651 1073 1406 1150">If an artifact was ever in state A, it should still be in state A</p>
 <p data-bbox="313 1444 548 1476">Guard Attribute:</p> <p data-bbox="391 1535 459 1566">$A(x)$</p> <p data-bbox="407 1587 443 1608">\Rightarrow</p> <p data-bbox="272 1629 578 1661">$\bullet \text{attribute} = \text{attribute}$</p>	<p data-bbox="651 1367 1406 1539">While the artifact is in state A, the value of the specified attribute(s) should remain unchanged. In other words, whenever the artifact satisfies $A(x)$, it should also satisfy $\bullet a_i = a_i$ for the specified attribute i</p>

Continued on next page

Visual Notation and LTL	Interpretation
<div style="border: 1px solid black; padding: 5px; margin: 10px auto; width: fit-content;"> <p style="text-align: center;">A</p> <p style="text-align: center;">=</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;"> <i>attribute</i> </div> </div> <p>Invariant Attribute:</p> <p>• $attribute = attribute$ since $A(x)$</p>	<p>Once the artifact is in state A, the value of the specified attribute(s) should forever remain unchanged. In other words, if the artifact ever satisfies $A(x)$, it should from that point onwards, make the specified attribute invariant, i.e., the attribute does not change its then assigned value in subsequent history entries.</p>

Table 3.1: Basic path restrictions and their equivalent temporal assertions. Note that each state has an associated boolean function specified over the attributes of the artifact type. In our notation we use shorthand notation $S(x)$ to denote the application of the function of the state S to the attributes of the artifact x .

The notation presented above gives a formal meaning in CLTL to each of the diagrammatic conventions used in specifying a DCML diagram. For example, the following augmented version of the two state DCML diagram presented in Figure 3.4 embeds two additional constraints using the notation presented in Table 3.1:

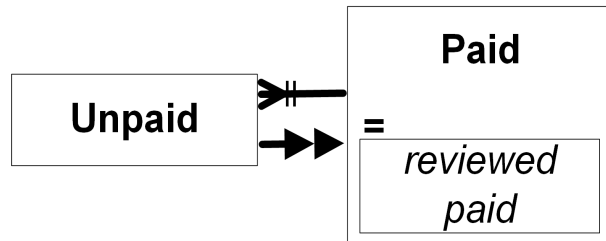


Figure 3.5: A more complex set of restrictions between the Paid and Unpaid States

In addition to the constraint present in the original DCML diagram (Figure 3.4), three other constraints are embedded in Figure 3.5. First, let us examine the double headed arrow ending in the Paid state, which imposes an entry restriction that can be interpreted from Table 3.1 as follows:

$$(Paid(SalesOrder) \wedge \neg \bullet Paid(SalesOrder)) \Rightarrow \bullet Unpaid(SalesOrder)$$

The above implication can be expanded using the definitions of Paid and Unpaid functions and is very similar to the one presented earlier for Figure 3.4. The net result is that now there is both an entry and an exit restriction between the Unpaid and Paid states, and these constraints must now independently hold true for all SalesOrder artifacts. Informally (in the business sense), we can describe the interplay of both these constraints simultaneously as follows: An artifact can only move from the Unpaid state to the Paid state and if an artifact arrives in the Paid state it must have come there from the Unpaid state. Having both an entry and an exit restriction between the two states makes the transition system much more rigid in its flow/direction.

In addition to the above discussed two constraints, there are two invariant constraints in Figure 3.5 that make the *reviewed* and *paid* attributes immutable when the artifact reaches the Paid state. The invariant constraint for the *reviewed* attribute can be interpreted using Table 3.1 as follows:

$$(\bullet reviewed = reviewed) \textbf{ since } Paid(SalesOrder)$$

or equivalently by expanding the definition of the Paid function as:

$$(\bullet reviewed = reviewed) \textbf{ since } (paid \wedge reviewed)$$

Similarly, the diagram also requires that the *paid* attribute also is invariant (unchangeable) when an artifact reaches the Paid state. Note that because embedded in the state definition is the requirement that *paid = true* (i.e., if an artifact is in the Paid state its attribute *paid* will be *true*), a natural consequence of the above constraint diagram is that if an artifact ever reaches the Paid state its *paid* attribute will remain true from that point

onwards in all history entries.

We summarize this section by emphasizing that our proposed database constraint modeling language tries to mimic the notion of states and path constraints as they exist in classical workflow and process modelling languages. However, because the language is formally related to CLTL, it can produce complex visual models that may be difficult to understand for the reader not familiar with temporal logic. Furthermore, we do not make the claim that the visual-CLTL notation that we have presented covers all possible LTL style flow constructs. However, it will serve as the basis on which generic visual-CLTL mappings can be established by policy makers specializing in various domains. We shall discuss improvements and extensions to this notation in subsequent chapters and propose various fragments of DCML for specific situations. However, the above proposed DCML notation should suffice for now in explaining and highlighting the benefits of database level visual constraint modeling in general.

3.5.2 Artifact Lifecycle Constraints

Initialization Constraints

To support the notion of constraining the start of artifact lifecycles in a database visually, we denote the non-instantiation of an artifact using a special state ϕ_S . Transitions *from* the state ϕ_S into any other state are considered as initializations, and therefore they are used in our framework to define the function $\phi_S(A)$ which, as mentioned in Section 3.4.4, needs to be interpreted at $t=0$. Furthermore, we only allow multi-way exit restrictions *originating from* ϕ_S in our diagrammatic convention and interpret them slightly differently than the visual-LTL semantics of exit restrictions presented in Table 3.1.

Consider the two exit restrictions from ϕ_S presented in Figure 3.6. From the diagram we see that there are two exit constraints as follows: (1) if an object exits ϕ_S , it must enter state D, and (2) if an object exits ϕ_S it must enter either state B or state C (or both). Both these constraints must independently hold true. Since $\phi_S(A)$ is interpreted non-temporally at $t=0$, in our notation we simply combine these two constraints to define the initialization function $\phi_S(A)$ as the conjunction of all the individual exit restrictions

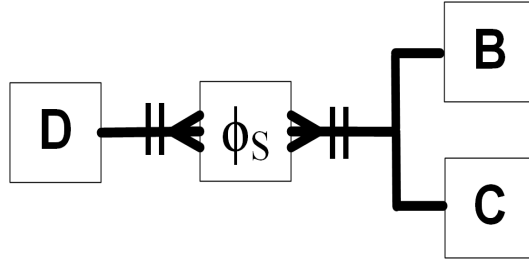


Figure 3.6: Two different exit restrictions originating from ϕ_S .

on ϕ_S . Consequently in the above example, $\phi_S(A) := D(A) \wedge (B(A) \vee C(A))$. In case the policy maker does not wish to restrict the instantiation of artifacts, s/he can choose not to use ϕ_S in the DCML diagram at all, in which case $\phi_S(A)$ will be implicitly defined as:

$$\phi_S(A) := true$$

and thus place no restrictions on the attributes of the artifact.

We can thus summarize the discussion of ϕ_S by pointing out that it is essentially the “uninitialized state” from which only exit restrictions are possible. Each of these individual restrictions must concurrently hold true, and we interpret them at $t=0$ only by using them to define the function $\phi_S(A)$.

Going off-diagram

In cases where an artifact can begin its lifecycle (A_0) in an unrestrained fashion, it is important to recognize that the notion of an artifact “existing” is different from an artifact being in a user defined state. In general, it is possible that the attribute values of an artifact at time i (i.e., A_i) do not satisfy the requirements of being in any user defined states (i.e., all state functions defined on the artifact type are false for the assignment of attribute values of A at time i). In such cases, provided that ϕ is false as well at time i , the artifact logically exists but is considered to be “off-diagram” (i.e., not present in any of the states defined on a DCML diagram).

Similarly, as artifacts can change values arbitrarily throughout their existence, they can

also go off-diagram at various points in time to possibly reappear back in DCML diagrams in a different set of states. Consequently the reader should consider the notion of a “user defined state space” as the set of values an artifact can take such that the particular assignment of attributes ensures existence of the object in at least one state. In many situations it may be important to constrain the behaviour of artifacts (i.e., their attribute values) such that they do not go off-diagram but conform to well established paths in their respective user defined state spaces.

Lifecycle termination

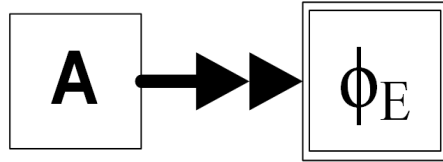


Figure 3.7: An entry restriction specifying that an artifact entering ϕ_E (the “dead state”) must previously have been in state A. The visual constraint can be interpreted by using DCML (Table 3.1) and the definition of the state ϕ_E as: $(\phi \wedge \neg \bullet \phi) \Rightarrow A(x)$

The notion of the “end state” for an artifact’s lifecycle is much simpler to describe. Such a state labelled ϕ_E (Figure 3.7) can simply be defined by the function ϕ . In other words, ϕ_E is a special state in which the lifecycle termination marker is set to true. Because the attribute ϕ can not be referenced by policy makers and we had implicitly required all attributes to become guarded invariants as a consequence of ϕ being true (Section 3.4.4), no special interpretation is required for this state for any CLTL constructs introduced in Table 3.1. In other words, notwithstanding the definition of the state function associated with ϕ_E , it can be treated as a regular state for constraint modeling purposes.

Policy makers can add entry restrictions towards ϕ_E as constraints by which an artifact’s lifecycle can terminate (i.e., delineate situations in which an artifact can enter the end state). Figure 3.7 is one such example where an object’s lifecycle can only be terminated provided that it was previously in state A. Because of the implicit constraints on ϕ_E introduced in Section 3.4.4, any other DCML constructs applied on this state will be

trivially always true, as the implicit constraints will never allow those constraints to be relevant. For example, exit is not possible from ϕ_E because (a) the termination marker cannot be unset, and (ii) the artifact can not change in any way. Thus, an exit constraint from ϕ_E is never relevant because the left hand side of the CLTL assertion/implication for such a constraint is never true.

3.5.3 Example

A more complete state space diagram that mirrors the lifecycle of a SalesOrder could be created by a policy maker as depicted in Figure 3.8. The standard method of implementing such a constraint diagram is quite simply to reject database transactions that violate any one of the five constraints (labelled 1 through 5 in Figure 3.8) for any artifact history.

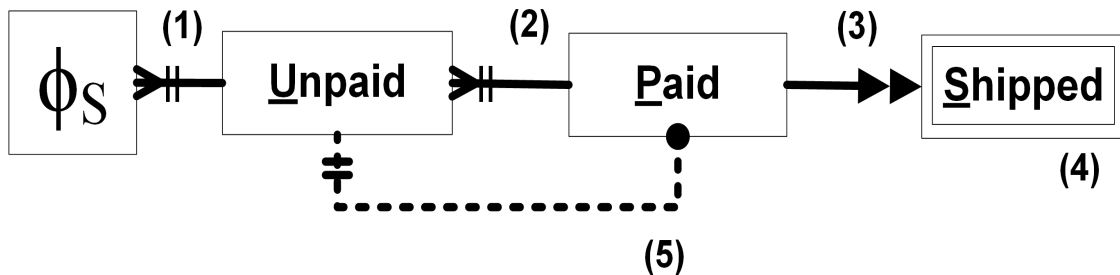


Figure 3.8: A set of restrictions relating and stitching together multiple states of the Sales Order artifact type.

Note the idea of stitching together states to define the proper flow in a process carries from the graphical/visual descriptions of business processes into DCML diagrams. However, there are many differences that arise when protecting illegal database state space traversals. For example, “going back” to previous states is implicitly not allowed in process models and business workflows (see constraint (5) in Figure 3.8). However, such implicit restrictions need to be made explicit in DCML. The next subsection examines various such issues and challenges posed by employing DCML to construct database level constraints.

3.5.4 Workflow Modelling in RDBMs

One of the benefits of looking at constraints as restrictions on the state space of database views is that there is often a very obvious and direct mapping from the business state space to the database state space available to be exploited. Database programmers and administrators do not need to spend countless hours modeling and examining constraint systems on their own, but can very well look directly at workflows and process models on which business managers rely. In fact, when using pre-existing workflows as the drivers for embedding constraints in database systems, there are two tasks of significance that need to be done: (a) establish a view oriented definition of the artifact being examined, and (b) identify how the attributes of an object in the view change as a business process progresses forward.

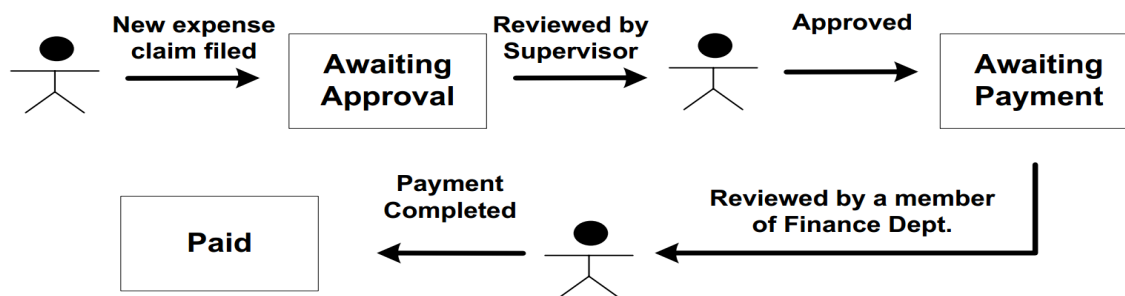


Figure 3.9: A hypothetical process model / workflow for expense claim processing. There are several similarities between a business oriented diagram and its equivalent for database systems presented in Figure 3.10.

The problem of identifying policy-centric views whose state space matters to policy makers is essentially the same problem as object relational modeling. Figures 3.9 and 3.10 provide an example of this naturally occurring correlation between workflows as envisioned by business users and DCML diagrams. The fact that in the context of business workflows the two diagrams are similar is no coincidence, because they should have the same states and aspects of flow (progress) among them. Since the DCML notation is relatively straightforward for many users to comprehend, we believe that once a definition of the artifact is agreed upon, business users themselves can visually create and edit these database constraint diagrams without the need for input from the database administrator.

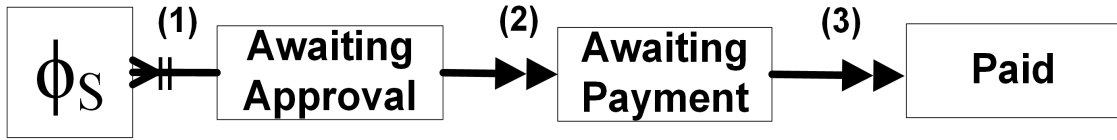


Figure 3.10: A database centric policy model with LTL constraints defined between the state of an expense claim view. The diagram corresponds to the workflow presented in Figure 3.9. Each labelled transition (1,2,3,4) represents a unique constraint specified in LTL that all objects in the view must adhere to. In a typical implementation, any transaction that violates any constraint for any artifact will immediately be rejected or rolled back.

3.6 Complex Workflow Construction

3.6.1 Multi-state Flow Constraints

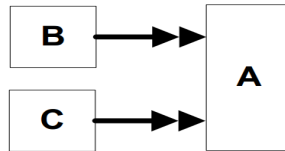


Figure 3.11: Two constraints on an object implying that if an artifact/object reaches state A, it must have previously been in states B and C (simultaneously)

One of the advantages, as well as a source of analytical complexity, of our modeling language arises from the fact that objects can enter and exit multiple states asynchronously. Figure 3.11 gives an example of a constraint diagram in which an object is required to trace multi-state paths. Note that the two temporal assertions derived from the model, $A(x) \wedge \neg \bullet A(x) \Rightarrow \bullet B(x)$ and $A(x) \wedge \neg \bullet A(x) \Rightarrow \bullet C(x)$, both must independently hold true. Also note that these two constraints indirectly apply an interesting subset relationship/constraint between states B and C requiring that objects reaching state A must previously be part of the implicitly defined state $B \cap C$. Although there may be situations where such implicit constraints can be difficult to comprehend, we will shortly show that this independent enforcement of each temporal assertion has significant benefits in policy management.

3.6.2 Multiple Paths and Progression

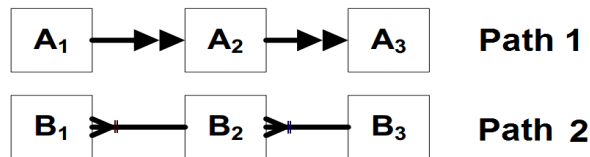


Figure 3.12: Multiple models of how an object can behave during different phases in its lifecycle can be independently created and temporal assertions across them can be enforced independently of other models.

Another benefit of allowing objects to be in multiple states at the same time is that different stakeholders' in an organization can freely describe their own relevant portions of the object lifecycles. Furthermore, they can do so without concerning themselves with the definitions of states used by other policy makers. For example, let us consider Figure 3.12 and observe that based on the constraints present, given a single update operation, an object can fail (or begin) to satisfy conditions to be in different states across both path 1 and path 2. More specifically, an object that is initially in states A_1 and B_1 , can after an update, not change state configuration at all or make progress according to the constraints in either or both paths. Each of these paths could have been created on a fresh clean canvas by a different policy maker, without the knowledge of the other. Since each constraint on each canvas is independently enforced on the history, many different DCML diagrams specifying constraints on a single artifact type can either be viewed together or independently.

3.6.3 Decentralized Workflow Modeling

The ability of objects to traverse multiple paths at the same time allows us to give a much broader meaning to the notion of an object lifecycle. Figure 3.13 presents an extended version of our running example of an expense claim. The diagram illustrates how three different departments of a business might have specified three different constraint diagrams on the same object. The figure shows that (1) the data entry staff must ensure that newly

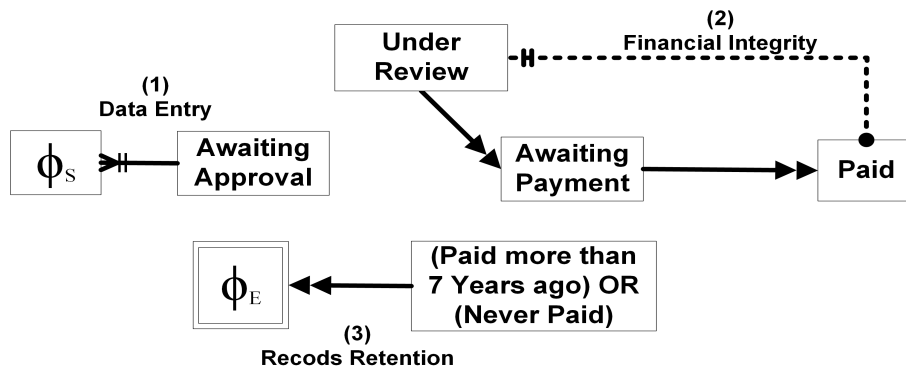


Figure 3.13: Individual constraint diagrams do not necessarily need to model all aspect of an objects lifecycle. The figure shows three separate constraint diagrams each accomplishing different objectives.

created expense claims are always in the Awaiting Approval state, (2) all claims that are deleted must previously have been in existence for seven years or never paid, and (3) financial integrity of the document (i.e., approval before payment and not marking a claim as unpaid after payment) is maintained. In short, Figure 3.13 represents the situation where three different DCML canvases on a single artifact type are viewed together on the same diagram. The different components of a broader state system show that we can model various intermediate phases in an object lifecycle without (necessarily) attempting to combine them in a single diagram. Thus, different policy makers can start constraining the same artifact in the context of different policies without having to consider all possible constraints imposed by other policy makers.

The flexibility of drawing out these constraint diagrams in a distributed fashion allows different policy makers to take charge of different aspects of constraints. Their roles can, of course, certainly overlap; for example, 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 advancing the expense claims from Awaiting Approval to Under Review states is handled elsewhere by another department of the business.

Single universal DCML diagram

In our formal model we have defined an artifact lifecycle as the sequence of values taken by an artifact (from $t=0$ to when the attribute ϕ becomes true). Each artifact of a given type has one such lifecycle, and various policy makers can arbitrarily define their own states for an artifact type and then enact restrictions between these states so that all artifact lifecycles will comply with these restrictions. However, we must emphasize that the notion of each policy maker having their own DCML canvas/diagram is merely a helpful illusion. This is because all constraints specified on an artifact type are equally applicable on all individual artifact histories. The fact that constraints are separately imposed on states defined on two different DCML canvases has no bearing on their logical meaning over the history.

Consequently, there is no difference between bringing the individual disconnected components/canvases onto one diagram (as done in Figure 3.13) or viewing them separately. In a manner of speaking, one can argue that there always exists only one unified (integrated) DCML diagram for every artifact type defined in a database of which different segments or “components” can be created/viewed by different policy makers responsible for that portion. This unified DCML diagram is essentially the union of all the different canvases used by various policy makers to define states and constraints on a given artifact type.

There are many benefits to this kind of flexibility, and foremost among them is that of simplification and reduction of the state space into a more component oriented view. Just because each policy maker *can* define their own states does not mean they always have to. Several policy makers can share states and their definitions, so that they can branch from one point in their lifecycle to a point in the others’ lifecycles. There are, of course, difficulties that can manifest in this situation, and much of Chapter 4 is devoted to discussing problems such as state overlap, state (un)reachability, and conflicting constraints.

Unified Instantiation and Termination

Finally a point that should become apparent by now is that, since an artifact lifecycle is instantiated ($t=0$) and terminated only once, the special states ϕ_S and ϕ_E have the same

definition regardless of their presence in various graphical sub-components of the unified DCML diagram for an artifact type. In other words, every policy maker who wishes to use either of these states to constrain artifact lifecycle instantiation and termination uses (shares) the same universal interpretation and definition of these states.

3.6.4 Generalized Path Constraints and Sub-formulas

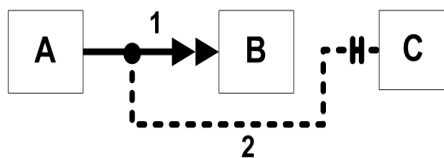


Figure 3.14: Constraint 1 being used as a state or a condition that needs to have happened in the past.

A common requirement in enforcing workflow policies is that of specifying conditional paths that require reasoning about a sequence of events. For example, consider the constraint that an object should never reach state C if it has sometime in the past reached state B directly from state A. Since this is a constraint involving three different states, it is impossible to draw it out as a ‘direct link’ between two states.

Graphically we propose that such constraints be modeled as reifying a path as a state. Observe that these “transitions,” as introduced in Table 3.1 on page 65, are all rules specified between two states, however we can also consider these rules as general requirements that must have happened in the past to invoke other rules. Consider constraint 2 (C_2) in Figure 3.14 as an example. It uses constraint 1 (C_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 never reach state C. More formally, (C_2) specifies the following temporal assertion:

$$(\blacklozenge C_1) \Rightarrow \neg C(x)$$

and since (C_1) is defined as $B(x) \wedge \neg \bullet B(x) \Rightarrow \bullet A(x)$, we get:

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

This elegance of linear temporal logic allows the use of a conditional assertion as a sub-condition to create larger, more complex assertions and to repeat the process by chaining it as many times as necessary to create conditional paths in policy-centric workflows. Furthermore, it allows us to easily go back and forth between complex assertions in temporal logic and equivalent graphical representations (constraint diagrams) without losing any information.

3.6.5 Link between Graphical Constraints and LTL

Assertion 3: *A constraint diagram can be converted into a set of assertions in first order constraint linear temporal logic.*

Because each aspect of our proposed DCML notation has a formal meaning in CLTL over the artifact history, policy makers can also see their constraint diagrams as a simple list of their temporal assertions of the form *pre-condition*(x) \implies *post-condition*(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 translate complex business workflows that are pertinent to their business functions into portions (fragments) of DCML diagrams. Whether a single policy maker chooses 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 are to be accomplished. In the next few chapters we exploit the above close association

between CLTL and constraint diagrams and examine in detail the framework for reasoning and implementation that it provides.

3.7 Related Work

In this chapter we presented a method of translating business process-centric workflows onto a state space defined by a set of functions over a database view. The use of a state space to describe possible outcomes (runs of a system) is a well known technique in computer science (e.g. Finite State Machines, Kripke Structures, etc.). The relationship between database state changes and changes in real world business situations has been acknowledged by researchers in various communities. For example, several prominent database textbooks have referred to the correspondence between reality and the database when discussing the closed world assumption. Similarly, much of the work done in the context of ECA and workflows is fundamentally grounded in the notion that events happening at discrete times move the system from various start states to some end states. The relative novelty of our approach lies in defining such a state space for elements of a database view. Relational views provide a very flexible abstraction for examining the contents of a database and thus can be tailored to accommodate reasoning over a wide variety of business processes.

3.7.1 *Declare* Workflow System

As discussed earlier, many researchers have proposed the use of LTL based modeling languages for constraints. However, the most similar modeling approach to ours is that taken by the *Declare* Project [59, 101, 161, 120]. *Declare* is a workflow modeling environment that also proposes a visual LTL notation for restrictions between different stages (tasks) in a workflow. The most significant differences between *Declare* and our proposed notation arise primarily because of the differences in target audiences between generic workflow modeling and monitoring systems and relational database systems.

Declare takes a broad, but simplistic, view of LTL constraints in workflows. States in

Declare do not necessarily represent the data contained in artifacts but can mean ‘tasks’ that may have restrictions among them. An example of a feature that our approach does not explicitly support³ is that tasks A and B could have a “not co-existence constraint” in *Declare*, meaning that they should never both be done. Being a general workflow modeling system, *Declare* does not restrict itself to specifying constraints using the past behavior of system and users can specify constraints about the future that resemble liveness properties (“something must eventually happen”). This is in contrast with constraints within database system that must always be enforceable on finite traces of a state space history.

There are several other differences arising from a database centered approach to visually depicting LTL constraints that, in general, business oriented workflow diagrams cannot accommodate. One such feature not present in *Declare* is that, in addition to an artifact being in multiple states in multiple lifecycles, it can actually go “off-diagram.” If an artifact does not satisfy the condition to be in any user defined state, then it still exists but is not part of any state on any specific diagram. Note carefully that the notion of going “off-diagram,” i.e., not being part of any user defined states, is different from object deletion or lifecycle termination. In DCML, artifacts can “disappear” and “reappear” at different states that are unconstrained, thereby defying traditional notions of workflows. Similarly, the states ϕ_S and ϕ_E are interesting anomalies that do not comply with the classical notion of an initializer or a start/end state in workflows that *Declare* attempts to model. Finally, we note that our proposed modeling language achieves significantly more extensibility than *Declare* through generalized (conditional) path constraints, as discussed in Section 3.6.4.

3.7.2 Integrated Data and Constraint Model

One of the major differences between our work and prior attempts is that we have essentially made the data model and constraint model inseparable. In order to circumvent the problem of establishing object-to-table mappings [52, 51], our model requires policy level objects and

³Yet. By the end of Chapter 6 the reader will hopefully be convinced that it is easy to come up with visual/graphical semantics such that our proposed technique of mapping LTL constraints can go beyond state oriented constraints.

artifacts to be declaratively defined on the underlying database schema. In fact, policy relevant business objects can only be defined as a logical mapping: that is, a business artifact in our model is defined as a tuple in a relational view over a database with a fixed schema. Our motivation for this is 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 [14]. Consequently, any reasonably powerful policy modeling system that is able to express constraints over complex business artifacts stored in a database needs to accommodate not only the underlying database schema (mapping), but also the actual data definitions and the queries used to generate these records.

3.7.3 Artifact Centric Process Modeling

Even in a mature field such as process modeling, we continue to see significant activity and major contributions. In recent years, increasingly many researchers have accepted that workflow style (process-centric) models for processes should be only a part of a much larger artifact-centric or data process modeling paradigm [109, 56]. Whereas traditional process modeling techniques focused on a process, such as online-ordering, and attempted to define its properties and lifecycle, the artifact centric process modeling paradigm attempts to aggregate many of these processes under a single umbrella denoted as an artifact. Although there does not exist a formal definition of an artifact, it can be considered as a collection of any and all data related to a particular entity or goal of an organization. For example, a customer, patient-visit, or Fedex delivery could be artifacts within some enterprise. From a business modeling perspective, the strict definition of an artifact is rather irrelevant, as any information that is needed as part of the tasks associated with the artifact can be included in it, and the segregation of data among artifacts is a business level decision left for the managers [135, 109]

Our research adopts a similar view of the world when looking at database systems. We consider everything relevant to a business (or a set of business policies) as a business artifact. Furthermore, we believe the best interpretation of an “artifact” in a database system is as a logical view defined over a database schema. Depending on users’ needs,

this view can be tailored to the task at hand: it may strictly define a single tuple used for policy-centric decision making or define several policy relevant objects concurrently. Observe that just as artifacts have no real boundaries, multiple views can also overlap (partially contain) each other. Similarly, we find it convenient to denote a single business artifact as one particular logical view in a database system, even though we could argue that an artifact could be the intersection, union, or cross product of many different and even unrelated views. We believe that our work largely complements the modeling efforts of the BPM community and offers a path to implementation of business constraints related to artifacts and their specific lifecycle components within database systems.

3.8 Summary

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 chapter we proposed a modeling language that strikes a balance in this spectrum, with its goal being seamless automatic generation of ECA constraints in a database as well as ease of use for policy administrators.

Our contribution is not simply to offer yet another business level rule modeling language. The existing process modeling techniques, such as state transition diagrams, flowcharts, data flow modeling, and workflow modeling, are sufficiently adequate in their own context. However because of the lack of a modeling language specifically for database systems (i.e., one that restricts the database state space or manages flow/progression of artifact behaviour), we present our own visual-to-CLTL notation called DCML. The most significant benefit of using this intermediate language is that, instead of directly constraining the database in SQL, policy experts (i.e., those who are not DBAs) familiar with a minimal degree of temporal logic can now convert high level workflows into artifact level constraint systems in the database.

In subsequent chapters we make the case for our modeling language being sufficiently

expressive to encapsulate most business process modelling techniques. Chapter 4 will discuss how these visual-CLTL models can be directly implemented as temporal integrity constraints in practical database management systems. We will also present a much more complex case study that exemplifies how a real life process can be converted into a coherent DCML diagram. Most importantly, we will also demonstrate how common workflow patterns can be accommodated in our database-centric policy modeling technique so that business users can easily convert their pre-existing workflows into database level workflows.

Chapter 4

From Specification to Implementation

In this chapter, we examine how constraints specified in DCML can be implemented in an off-the-shelf relational database system. The work presented builds on the comprehensive examination of issues pertaining to the efficient implementation of LTL constraints done by Chomicki [33] and Toman and Chomicki [154, 34]. Our analysis shows that in most SQL-99 compliant commercially available RDBMSs, an efficient implementation of LTL constraints requires no changes (source code modifications) to the database engine.

We present synthetic benchmarks and results of our implementation of CLTL constraints, showing very little compromise in transaction processing efficiency and minimal storage overhead. Characteristics of real world DCML diagrams that impact how efficiently certain constraints can be monitored in database system are also examined.

Our results support the prior research in this area and show that, even when business artifacts are part of an exorbitantly large number of concurrent workflows (making them subject to an extraordinary amount of constraints), a modern RDBMS can handle heavy transaction loads over such artifacts without a noticeable performance penalty in transaction processing. We conclude this chapter with suggestions for optimization of DCML constraint systems, including guidelines that can help system administrators create and monitor an efficiently implementable set of constraints.

4.1 Framework for Monitoring State Changes

4.1.1 Terminology

Before discussing implementation and efficiency concerns, we introduce terminology that will allow us to measure and benchmark the types of constraint systems generated by DCML. For a particular artifact type V , we denote $S_V = \{S_1, S_2, \dots, S_N\}$ as the finite set of all user-defined states for artifacts of type V . Recall that each of S_1 through S_N represents a function that returns true or false for an assignment of an artifact’s attribute values at any given point in time. We also know that at any given point in time, an artifact may belong to multiple labeled states if it satisfies more than one of these state conditions. In order to determine which states an artifact x is in, we will denote the state configuration of x , $SC(x)$ to be an ordered binary string (or bit vector) of length $|S_V|$. A zero at position i in the state configuration of an object x implies that $S_i(x)$ is false and that the object does not belong to state S_i . Similarly a one at position i in the state configuration of an object x implies that $S_i(x)$ is true and that the object belongs to state S_i .

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 attribute values of the object. Note that the set of possible state configurations for any artifact is always finite ($2^{|S_V|}$). We can better observe this in an example, where S_1 , S_2 , and S_3 are the only states defined system-wide for policy modeling on an artifact type. For this three-state scenario, the state space is denoted by a set of binary triples: $\{(000), (001), \dots, (111)\}$. Recall 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 in S_V (i.e., “off-diagram,” as discussed in Section 3.5.2).

It stands to reason that the more complex a description of a process model is, the more stages/states it will have. As a consequence “longer” and more complex business process models, when translated into DCML diagrams, will also have a higher number of states and possible state configurations ($|S_V|$ and $2^{|S_V|}$ respectively). Thus, this abstraction allows us to capture a dimension (i.e., number of states) across which we can benchmark the implementation of DCML models in database systems, and relate it more generally to

workflow length and complexity.

In order to reason over various state configurations of an artifact during its lifecycle, we define an artifact's *state configuration history* (SCH) as the time-stamped sequence of state configurations in which it existed. If we denote the state configuration of an artifact at time t as $SC(x_t)$, then the state configuration history is the ordered set $\{(t_0, SC(x_1)), (t_1, SC(x_2)), \dots, (t_c, SC(x_c))\}$, where c is the most recent (current) time-stamp.

The state configuration history is similar to the history of attribute values of an artifact. However, it comprises a single binary field whose length is equal to the number of user-defined states, and therefore it is very likely to be much more compact than the artifact definition itself. Given the semantics defined for DCML diagrams in Table 3.1 on page 65, a complete history of artifact values is never explicitly required for constraint enforcement. This is because constraints specified in DCML do not concern themselves with the actual attribute values an artifact took in the past but instead look at the consequences (resulting state configuration) of changes to those attribute values. Attribute values, when temporally referenced in DCML constructs (as guards and invariants), are only interpreted on the current and next time steps in the history. Therefore, maintaining a detailed history of attribute values that an artifact took in the past does not provide any benefits. Instead it is sufficient to store only whether a state condition was met or not at a given point in time in the past to enforce constraints. In other words, instead of physically storing the entire artifact history, which can contain large and complex attributes (strings and integers), we only need to store a compact bit-vector like representation of the policy relevant conditions (states) that an artifact satisfied at each time step. This space savings has significant consequences, which will be examined shortly.

4.1.2 Maintaining Artifact Histories

Since artifact types are defined as views, the constituent values of a single artifact can be stored in multiple base tables in a database. In this section we describe how a history that is spread across multiple source tables can be captured and maintained. We do so with

<u>INV_ID</u>	LASTUPDATE	CST_ID	AMOUNT	SHIPPED	PAID	APPROVED	...	NOTES
...
19846	13/05/12 14:27:04.244	764	59.95	NO	YES	YES	...	Shipping Delay
19847	29/05/12 22:18:05.263	589	129.97	YES	YES	YES	...	N/A
...
19986	17/05/12 01:25:35.849	798	299.97	NO	NO	NO	...	Awaiting Payment
...
...
19995	19/05/12 07:58:05.119	798	299.97	NO	YES	NO	...	Pending Approval
...

Table 4.1: A fragment of a typical invoice table: Each row uniquely represents the current status of an invoice. In this particular example we can see the contents of four invoices (19846, 19847, 19986, and 19995), all of which were last updated in May of 2012.

the following example, where a database has two tables, *Invoice* and *Customer*, and an artifact type as a natural join between the two tables defined as follows:

CUSTOMER: (int CST_ID, varchar NAME, varchar ADDRESS, ..., varchar ADDRESS_COUNTRY, varchar CST_TYPE)

INVOICE: (int INV_ID, datetime LASTUPDATE, int CST_ID, float AMOUNT, bool SHIPPED, bool PAID, bool APPROVED, ..., varchar NOTES)

InvoiceProcessing_ARTIFACT:

```
SELECT INV_ID, CST_ID, ADDRESS_COUNTRY, CST_TYPE,
SHIPPED, PAID, APPROVED
FROM CUSTOMER JOIN INVOICE
ON INVOICE.CST_ID = CUSTOMER.CST_ID
```

Tables 4.1 and 4.2 depict what an audit trail of invoices would look like and present an example of an auxiliary relation used for auditing purposes. For illustrative purposes we depict the full attribute level history of InvoiceProcessing artifacts. All new invoices (without any prior history) are directly inserted into the invoice table (Table 4.1). An auxiliary relation called InvoiceAudit is implicitly defined in the database to keep a log of changes to invoices (Table 4.2). If an existing invoice is modified (i.e., a row in the invoice table is updated) the original values of that row are inserted as-is into the audit table before the modifications to that row are persisted with the current time as the LASTUPDATE

<u>INV_ID</u>	<u>LASTUPDATE</u>	<u>CST_ID</u>	<u>AMOUNT</u>	<u>SHIPPED</u>	<u>PAID</u>	<u>APPROVED</u>	...	<u>NOTES</u>
...
19846	11/05/12 23:34:15.077	764	59.95	NO	NO	NO	...	Awaiting Payment
19847	12/05/12 08:34:05.235	589	129.97	NO	NO	NO	...	Awaiting Payment
...
19846	12/05/12 14:19:37.665	764	59.95	NO	YES	NO	...	Pending Approval
19847	12/05/12 16:57:03.187	589	129.97	NO	YES	NO	...	Pending Approval
...
19995	19/05/12 07:58:05.119	798	299.97	NO	NO	NO	...	Awaiting Payment
19847	21/05/12 16:57:03.187	589	129.97	NO	YES	YES	...	Shipping Delay
...

Table 4.2: A fragment of the “audit table” for the base invoice table (See Table 4.1). The rows represent older rows from the original table, therefore requiring an extended primary key. Such tables are append-only and only accessible to highly privileged users.

field. This mechanism ensures that the valid time of facts can be preserved by recording successive transaction times. A similar audit table for the customer table could also be introduced if changes to a customer’s data were significant in a business policy or there was some business need to audit past customer addresses. For example, it is conceivable that a customer changes address multiple times while his/her invoice is being processed, thus requiring special steps in the invoice processing workflow.

In many regulated businesses the continuous auditing and logging of row level database modifications is commonplace. The business applications, their users, and even programmers are often not aware of the database storing a history of changes through auxiliary relations, nor even the fact that such tables exist. Procedures to create these auxiliary relations and the relevant auditing *triggers* on base data automatically are widely adopted in the commercial use of databases [107]. It is also important to point out that the audit information may not be exclusively used for compliance but could also be used for analyzing and improving the performance of a business. For example, in this scenario audit information could be mined to determine the most significant causes for delays in invoice processing or the average wait time between payment and shipping. There are also space optimized ways of maintaining an audit trail, such as delta-encoding, which only stores the attributes that were modified in a given update. We defer their discussion to Section 4.3. However, we can summarize the key points of this subsection by noting that auditing

(keeping a history) of a complex artifact that is spread across in multiple tables is simply a matter of auditing its respective constituent tables and that data logging triggers are the method of choice to maintain audit trails.

4.1.3 Lifecycle Management

Since the InvoiceProcessing artifact is defined as an inner join between the Customer and Invoice tables, an audit-table for the InvoiceProcessing artifact is conceptually no different than an inner join of the two individual audit tables. In fact, even when an artifact is defined as a multi-way join among many tables, the requirement of uniqueness of individual rows therein provides us the ability to recreate the history of the artifact by combining the relevant underlying audit tables. The only caveat is that we now have to consider the latest of all LASTUPDATE timestamps in each constituent base table history as the last time at which the entire artifact was updated.

Separating histories of constituent tables brings about an interesting consequence in interpreting certain changes in artifact history. To elaborate on one such problem, let us consider the case where we have an invoice currently being processed and it is associated with a customer named Alice. Let us also say that Alice has changed her address several times, so she has a customer history and so does the given invoice. Now imagine that midway through processing this particular invoice, for some reason, it has to be assigned to a different customer (Bob). Figure 4.1 depicts this change happening at timestamp 4. If we consider the invoice alone, there is nothing extraordinarily special going on in its history, as only one attribute of the invoice (CST_ID) was changed. Similarly both the customers have their own relevant histories, which are completely unaffected. However, note that an artifact is uniquely identified by a pair of invoice and customer identifiers, which means that at time 4 the history of the *particular invoice as it was associated with Alice* (facts A, B, and C) is terminated. At the same time our model ensures that ϕ is set to true and the same unique artifact will forever remain in the dead state never to reappear. Also, at the very same time, a new invoice-customer pair is added, starting a new artifact history for the invoice as it is associated with Bob and thus beginning a new artifact lifecycle (D, E, F). In other words, an artifact in this scenario is an invoice as it

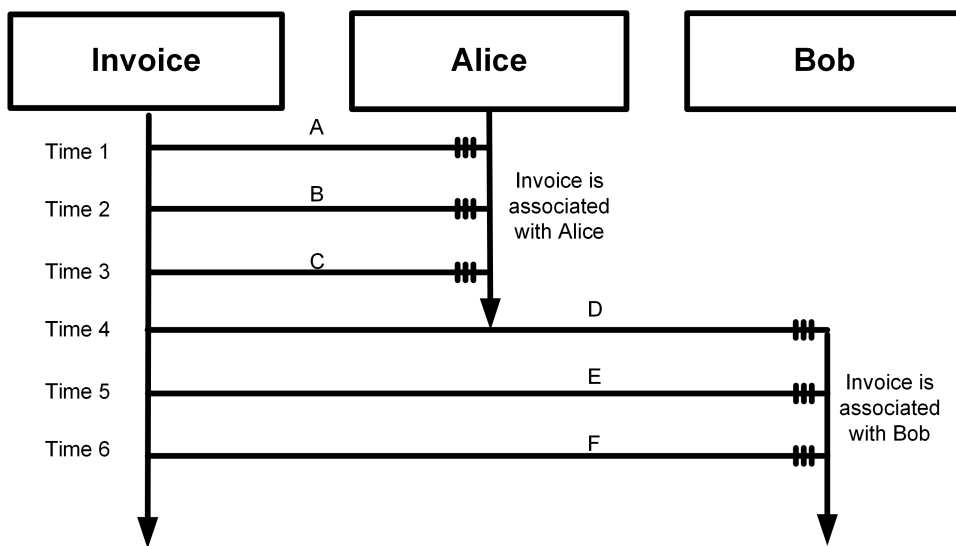


Figure 4.1: An invoice can be associated with one customer at any given point in time. Changing the associated customer has no significant consequence on the complete history of an invoice (A,B,C,D,E,F) but the old history of the artifact (A,B,C) terminates and a new one starts (D,E,F) at time 4.

relates to single customer, and one such artifact lifecycle terminates while a new one is instantiated at timestamp 4 (provided $\phi_S(A)$ for that artifact holds as well).

The key point in the scenario depicted in Figure 4.1 is that, even though there is only one invoice, it can interact in multiple artifact level lifecycles. Thus, business policy makers and DBAs catering for all possible exceptions in a business workflow should be aware of the intricacies of modifying data and its implications on the history of not only individual objects but the collective processes they participate in. In order to keep our example simple and straightforward, we shall assume that invoice numbers and customer numbers are assigned only once and cannot change over time. This assumption of unique object/artifact identifier is embedded as a requirement in the model as described in Section 3.4.3 and, when combined with the fact that each artifact has only one lifecycle significantly simplifies enforcement related issues. Finally, we point out that a state configuration history is only a layer of simplification on top of traditional auditing, and it too can be derived from the underlying audit-tables by simply evaluating the boolean state conditions over the recorded facts.

4.1.4 Materialization of State Configuration Histories

An alternative to continuously (re)creating the entire artifact state configuration history from the underlying base tables (at every update) is to materialize and maintain it. In other words, we can forgo keeping a physical audit trail of artifact values if we only store the audit trail of states that an artifact was present in. Database systems, such as IBM's DB2 and Oracle's 11g, offer direct support for triggers over views using the infrastructure developed for the incremental maintenance of materialized views, and these triggers can be used to log and determine the validity of all updates to an artifact's state configuration history automatically.

There are several major advantages to such offloading of the task of monitoring changes to artifacts (views) to the database engine and use the capability to trap (examine) and reject those changes. Foremost, we can exploit the facilities provided by the database engine to monitor views instead of writing complex triggers on the base tables to do so (this task is essentially made redundant). It can be argued that user-programmed triggers would be

far less efficient than those implemented and executed by a database engine to monitor materialized views, especially in a scenario where a large number of these overlapping artifacts (views) are defined for enforcing business rules. Since the database engine is well aware of the entire schema, it can better isolate irrelevant updates and therefore minimize the transaction processing overhead caused by the monitoring of artifacts. In fact, given a certain workload, modern database systems can provide guidance into whether materialization will have a positive impact on transaction processing [177], and they can even automatically materialize specific portions of base relations in anticipation of peak workload conditions [176].

Second and more importantly, we can choose to materialize the state configuration history instead of “recalculating” it at every decision point regarding A_{next} . That is, instead of continuously logging changes to base tables, we can (i) only log the changes relevant to specific artifacts and (ii) only log all changes relevant to the state configuration history of those artifacts. With regards to point (i), note that, since the state configuration history is itself a type of mini audit-log for artifacts, we can avoid auditing complete tables and only maintain history entries for specific artifacts subject to constraints. With regards to point (ii), materialization of the state configuration history only also avoids change tracking for irrelevant changes to artifacts. For example, an invoice may be physically updated, but not in a way that impacts any of the business states that it is in. Thus, from a workflow and constraint perspective, there may be no need to audit changes that do not impact the states an artifact is in.

Between these two options (materialization and derivation), there is of course a trade-off between extra storage space (involved in storing the state configuration history) and processing time (evaluating the boolean conditions to re-derive it). We will examine this tradeoff between methods of maintaining a state configuration history in Section 4.2, when we present our benchmarks for monitoring and maintenance of such histories. For now, we assume that a logically correct history for all artifacts is available for policy-centric decision making, and we note that mechanisms to monitor changes to database systems are commonly available to do so.

4.1.5 Checking Individual Constraints

Given the availability of a state configuration history for artifacts, the final piece of the puzzle lies in continuously examining it for policy violations. The key to enforcement lies in the observation that a temporal formula in CLTL can also be treated as a validation query over a finite history. Compliance of a database with a constraint in CLTL is thus equivalent to trapping transactions that change the state configuration of artifacts and checking whether the resulting change will lead to a sequence in which the negation of the constraint is satisfied. Transactions that do not violate any constraint are allowed to commit, and those that violate one or more constraints are aborted.

The method (algorithm) to verify and validate histories is straightforward. It relies on directly interpreting temporal operators (as described in Section 3.4.3) in first order logic and then issuing SQL queries (recursively) over the history to evaluate them. If the query returns true for a constraint / CLTL formula, then the constraint is valid on the history. Algorithms and detailed descriptions of their implementations (including operators to accommodate metric-temporal constraints) have been repeatedly presented in prior literature, (see [154, 34, 33, 75, 18]) and we omit their discussion from this thesis.

4.2 Measuring Policy Overhead

The most widely cited implementation of first order temporal integrity constraints implemented as SQL triggers, which reject (roll-back) transactions if a first order temporal integrity constraint is not met, was presented by Toman and Chomicki [34]. The work introduced the use of result memoization to check past temporal integrity without examining the entire history of an object. A simple example of memoization can be demonstrated with the condition $\blacklozenge S_A$. Determining whether sometime in the past a complex event happened can require us to trace the entire history of an artifact. Moreover, re-determining whether the same complex event has already happened at every update can slow down the process of constraint checking. In many cases it may be easier to store details about complex events in the past alongside the artifact. In other words, we can make the fact that “something has happened in the past (or not happened in the past)” an inseparable part of the object

in question, instead of rediscovering this fact by traversing the history every time an object is updated. Toman and Chomicki demonstrated that such a technique allows us to check constraints much more efficiently with at most a polynomial (in terms of formula length) amount of extra space. The performance results we now present augment the work done by Toman and Chomicki, and our work distinguishes itself by focusing on issues specific to the problem of implementing LTL constraints that originate from DCML diagrams. Furthermore we provide actual benchmark test results in the context of a modern database system.

There are two sources of overhead on which we focus in our tests. Foremost is the trade-off between space and time, that is, the design decision to store the state configuration history or to re-create it from the audit logs whenever needed. Second is the actual processing time for verification of constraints, including the time spent to check if an object is in a particular state or not. Our tests show that, in typical 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 will be negligible. In addition the processing penalty of checking typical business conditions on objects (state conditions) is almost non-existent, as the information needed to make policy-centric decisions is available in memory and relative to the actual cost of writing results onto disk, the cost of in- memory processing/checking of constraints is almost immeasurable.

For our tests we considered the business definition of a typical invoice or purchase order artifact as specified in the TPC-H benchmark to guide us. The Transaction Processing Council (TPC) publishes several open schemas and benchmarks that are well known in the database community. We used the TPC-H benchmark schema and tested against business scenarios of varying complexity by considering corporate workflow (pertaining to a single artifact) of size 128 states, 512 states and 1024 states. Since TPC-H does not describe specific business processes, these states were synthetically created. For our tests we used TPC-H databases of sizes 1GB and 10GB on an Intel Core 2 Duo based machine with 4GB RAM running Microsoft SQL Server 2008.

4.2.1 Storage Overhead

Motivation

Although having as many as 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 bits required by the state configuration history. Observe that the state configuration history is a compact representation of an object's membership in all user defined states. Even in our extreme case, 1024 true/false results are essentially 1024 bits of information (128 bytes), and this is still less space than a single 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 used, for example, for notes and comments. Consequently even when considering a corporate workflow of 1024 independent states with no exploitable similarities (i.e., the truth/falsehood of one state condition has no bearing on the truth/falsehood of any other), an extraordinarily large relative storage burden (the ratio of the size of the state configuration to the size of the actual object) will perhaps never be encountered in real life situations. More surprisingly, our test results showed that even when this ratio is taken to extremes by considering the consequence of having tens of thousands of states, a database under typical transactional loads will not suffer significant performance degradation.

Test Setup

To measure the impact of additional storage on the day to day operations of a transactional database system, we conducted several tests by adding variable length bit vectors alongside artifacts and measuring the time taken to complete typical database transactions (inserts, deletes, and updates). These bit vectors essentially represented different length state configurations, which in turn represented the number of states that were defined on an artifact, which in turn were a proxy for the complexity of workflows defined over such

artifacts.

Results

Our tests showed that, in general, appending additional binary fields to relations causes no performance degradation for random updates. In fact we were able to go well beyond our extreme case and reached 10,000 states before noticing any significantly measurable difference. The rationale behind there being no additional cost to tacking on a large bit-vector is clear in hindsight. A random update to a row in the invoice or line item table in the database causes only one page to be physically accessed and committed to disk. Thus, as long as the state configuration can be accommodated within the same page as the object being updated, the cost of writing/flushing this page to disk will not increase. Since the cost of fetching and writing a page is consistent in both conditions and also the most significant component of a transaction, there is no observable difference in time taken to complete a transaction. In most database deployments a page size of 4KB (32,768 bits) is common. We believe that even for the most complex workflows, this limit will never be reached, and the actual storage of the state configuration does not pose significant transactional costs.

Note that our objective was solely to test whether a system under a transactional load (high-update situation where a random invoice or line-item is impacted) is stressed by the additional overhead of writing the state configuration alongside the object. There are, however, situations where it might be preferable (for example, a non-random workload, sequential scans, etc.) to store the history in a completely separate database (i.e., an extension of the independent audit-trail option discussed in Section 4.1.4). This would provide the additional benefit of securely isolating the audit trail from the operational data. Furthermore, if the audit-database is on a separate disk then the performance penalty incurred for different types of workloads (mix between various types of read and write queries) can be better managed.

4.2.2 Computational Overhead

Motivation

For each transaction that modifies an artifact, we require that the new state configuration be calculated and then compared against the old state configuration(s) to ensure that all specified constraints are being met. We anticipate that in many situations where states for business level policy conditions share common variables, we will not have to incur the cost of checking every state conditional independently. For example, consider a workflow for invoices with two states called, paid and unpaid, with the conditions, “paid = true” and “paid = false” respectively. Observe that we need to check only one of these conditions 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 possible constraint violations dynamically. These optimizations aside (discussed further in Section 4.3), in our tests we took a pessimistic stance by assuming that there are no avenues of optimization available for the database engine/query compiler to exploit.

Test Setup

We conducted tests 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 used to monitor the implications of an update such as check constraints, triggers, and constraints on materialized views. Our tests were designed to explore the costs associated with repeatedly calculating the state configuration, and our objective in this section is to provide a reference for comparing the practical computational costs of two of these possible techniques in light of a varying number of assertions specified in first order temporal logic.

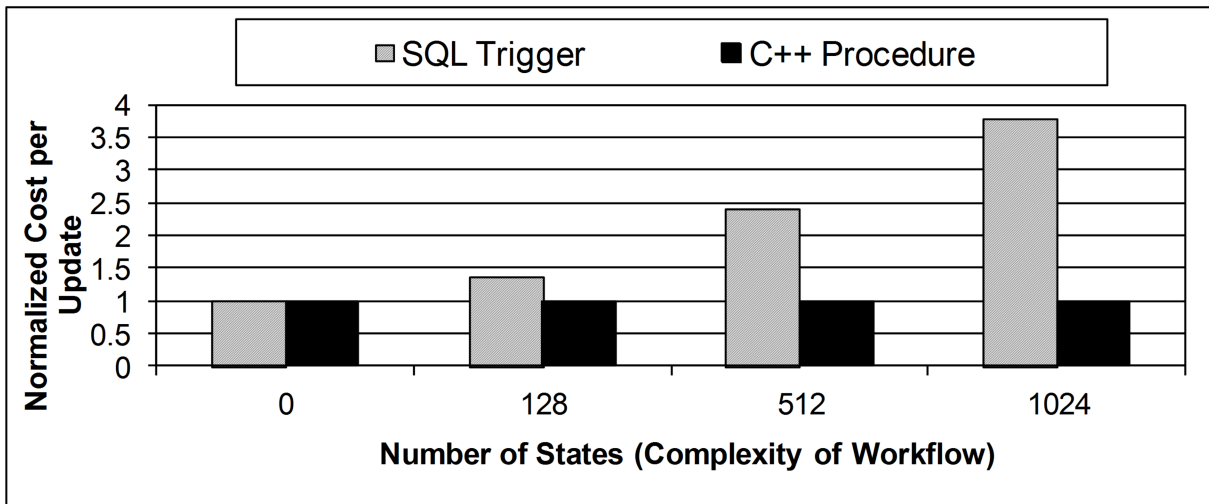


Figure 4.2: Time taken by a SQL server trigger to complete simple in-memory if-then styled checks grows linearly with an extremely high overhead per check. Very quickly these checks become the bulk/bottleneck in trigger processing. A C++ procedure however responds in a very predictable fashion where performing the exact same in-memory checks has no bearing on the total cost of the transaction.

Once again we used the number of states (one condition per state) as a proxy for the amount and degree of workflow complexity: 0 states (to represent no computational overhead as a measure of baseline costs), 128 states, 512 states, and 1024 states. Under this test setup, adding an extra state essentially means incurring the cost of an additional synthetically designed check to see whether an object belongs to that state or not. These checks were designed to simulate traditional business level string and arithmetic comparisons (such as “status = paid”, “shipcode = M”, “amount \geq 250”, and “shipinstruct=overnight”) 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. Thus, there was no scope for optimization.

Results

Our hope was that we would be quickly able to demonstrate a very obvious fact about calculating the state configuration history. That is, determining whether an object belongs to a particular state through a simple check (SQL if-condition), where all the necessary data to do so is already in memory, would pose no transactional cost. After all, simple one line statements (“status=paid” or “amount \geq 5”) on data that is already memory-resident, even when repeated thousands of times, should not be comparable to disk writes (which are several orders of magnitude slower than memory/CPU operations and dominate database transaction costs).

Our results (depicted in Figure 4.2) show that, contrary to what was expected, SQL based triggers, as implemented in Microsoft SQL Server 2008, are extremely inefficient at performing even the most basic operations on primitive data types. We observed a linear correlation between time taken to execute and the number of synthetically designed checks (essentially IF-statements in SQL to compare attributes against static values). At 1024 states Microsoft SQL Server took upwards of ten seconds to complete one invocation of such a sequence of checks. Our attempts to identify the cause of this significant slow down caused by increasing the number of in-memory operations was hampered by the lack of instrumentation provided by SQL Server to analyze the execution of triggers. Note that at 1024 checks our source code for a typical trigger was around 100KB, and we were able to verify that SQL Server was not re-compiling the trigger at every invocation. We were also able to confirm that the delay was purely in the execution of the compiled code (i.e., not waiting on disk or other resources). Our conclusion here is that for some reason the compiled code being generated for trigger processing was far more inefficient than expected. These results were not unique to Microsoft SQL Server and similar results demonstrating poor execution performance of SQL code were found when using IBM’s DB2 database server.

Fortunately, most database systems (including Microsoft SQL Server) allow the execution of externally written procedures, that is, executing a memory-resident program outside the database engine, and exchanging relevant parameters for analysis. Programs in a language such as C++ can be called upon to do any form of work if the database/SQL

execution engine is not adequate¹. Such techniques are commonly used in scientific applications, where external programs such as MatLab and R are used to perform data analysis and store the results back in a database. When such an approach is adopted (moving the if-conditions into an external C++ program), there is no discernable increase in the time to test 1024 state conditions than to perform no tests at all.

We conclude that although methods for automatic generation of SQL level triggers have been well examined in prior literature, implementing them efficiently (i.e., in a manner that does not immensely slow down the system) is anything but straightforward. Because of the vast differences in SQL execution behavior and how different database engines handle trigger code, an LTL constraint to SQL trigger generator will need to be optimized for each individual database product. That being said, this customization will require little effort but a lot of familiarity with the performance quirks of a given database engine.

4.3 Optimizations

During the course of this Chapter, we briefly discussed several sources of optimization and cost savings. In this section we provide more details on how they can be practically used and discuss their overall benefit.

4.3.1 State Space Reduction

Using LTL constraints 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. The most obvious optimization through static analysis comes from eliminating unsatisfiable states. An unsatisfiable state is one which an object can never attain by virtue of the definition of the state alone (that is, no assignment of the attributes of an artifact can satisfy the condition of the state). The problem of determining satisfiability of a state is equivalent to solving the underlying SMT/SAT problem on the

¹either using a call to pre-compiled memory resident DLL (extended stored procedures) or the interface provided by the xp_cmdshell call in SQL Server: see [150, 87] for reference and usage

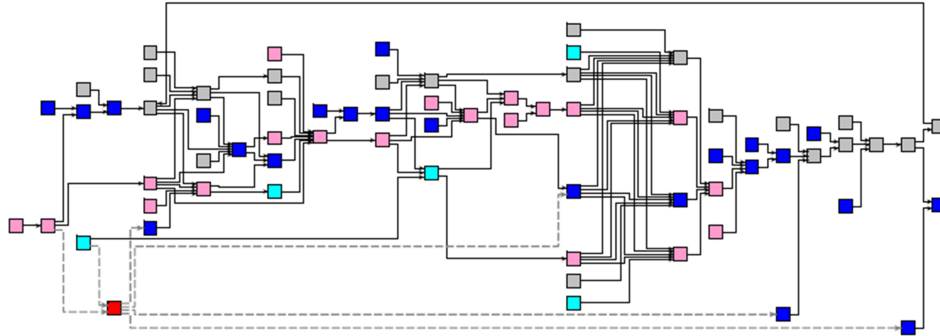


Figure 4.3: Different users may be interested in different paths of complex workflows (shown in different colours). States where these paths intersect may need to be negotiated together, jointly defined, or at the very least collapsed together to make enforcement more efficient.

attributes of the artifact. Note that the cost of determining whether each state is satisfiable has to be incurred only once during static analysis of the constraints and can save significant time in repetitive checking of an unsatisfiable state condition during run time.

Furthermore, an unsatisfiable state most likely represents an error on the part of the users of the system, as they have no use in constraint modeling and thus should not be defined. It is perhaps wise to report such states to the policy maker so that s/he can resolve any errors associated with the incorrect assumption that the state under question is meaningful for policy purposes.

Similarly, merging states that are “equivalent” across different models (sub-components of a unified DCML diagram) created by different policy makers can also be a significant cost saving measure (see Figure 4.3), where states are *equivalent* if and only if the same set of attribute values cause their respective functions to be true and false. In other words, the associated state functions of two equivalent functions are rewrites of each other. Once again, demonstrating the equivalence of two arbitrary constraints specified on a set of (non-boolean) attributes is not trivial but can under many circumstances (e.g., finite domains) be reduced to the problem of classical boolean/propositional equivalence.

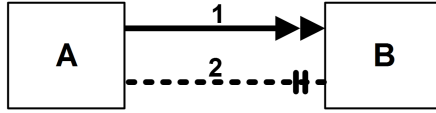


Figure 4.4: Temporal inconsistency (dead end).



Figure 4.5: An indirectly specified containment restriction.

4.3.2 Constraint Analysis and Sanity Checking

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 4.4. Constraint (1) mandates that all objects exiting state A 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 transitivity we get, $\bullet A(x) \Rightarrow \neg \blacklozenge A(x)$, interpreted as, previously A implies never in the past A, which is an inconsistent (unsatisfiable) temporal formula. What this means is that the transition to B simply cannot take place and neither can the object leave state A. Thus state A could potentially represent an unintentional dead end in the workflow for the object. We hypothesize that such inconsistencies will emerge as the result of human error, poor policy planning, or conflicting policies.

Note that the problem demonstrated in Figure 4.4 (dead end) has no bearing on the actual definition of the states A and B. More specifically we did not have to look at the definitions of $A(x)$ and $B(x)$ to come to the conclusion that A cannot be exited. It is merely the way two constraints interact that an interesting result about a workflow emerged. Furthermore, the fact that $\bullet A(x) \Rightarrow \neg \blacklozenge A(x)$ is unsatisfiable is trivial to see visually in this diagram. However, as large diagrams with hundreds of state and constraints are integrated, the detection of these abnormalities becomes challenging. In subsequent chapters we will

demonstrate that LTL satisfiability solvers, armed with guidance on what to detect, can play a role in significantly reducing policy and reachability errors by considering hundreds of temporal implications concurrently. The appropriate results in many cases can help simplify constraint diagrams.

In fact, additional reasoning about constraint-oriented workflows can play a significant role in simplification of constraint systems. The overall objective of analyzing constraint interplay is primarily to identify “interesting” facts about logically correct constraint systems that can lead to avenues of optimization in constraint-oriented workflows. An example of such analysis is presented in Figure 4.5, where 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 interpret 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. Therefore, objects in state B must have previously been in both state A and state C at the same time. This of course raises several questions related to containment of states, that is how closely is being in state A related to being in state C? Are either of those states fully contained in the other? Furthermore if the analysis of the conditions of A and C shows a contradiction (for example $A \cap C$ being empty) then we may have a problem in the form of a logical inconsistency, that makes state B unreachable.

Two important observations about logical reasoning that concerns 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, during this constraint integration phase, 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 restricted by many factors, including the view (object) definitions, domains of the attributes, func-

tions used to define states, etc. For arbitrary queries, domains, and state conditions, the problems of determining containment, closure, and intersection are well known to be undecidable. However, we believe that a significant number of business object definitions will involve the use of simple conjunctive queries and operators that are relatively simple to reason with. 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 more comprehensive discussion of the computational cost and tools available to optimize/model check a given set of assertions in first order logic is presented in Chapter 5.

We conclude this section by pointing out that these optimizations are one-time costs and in typical operations need to be incurred only when a new constraint is introduced in the system. Furthermore, we argue that it is unlikely that new individual constraints will impact all known properties about a complete system, and thus the scope of analysis with new constraints will require minimal incremental analysis for optimization.

4.4 Summary and Related Work

4.4.1 Active Database Systems

There are many aspects associated with the development and use of active database systems that have over time been encapsulated and abstracted by modern database engines. The event monitoring and view maintenance facilities provided in a modern DBMS, for the most part, hide the complexity associated with building an efficient event detection infrastructure. Cochrane et al. [38] have provided a very detailed description of how triggers and events are implemented in relational database systems. Their work summarizes the trigger related features offered by most commercial DBMSs and provides an overview of their implementation. 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. Many active database prototype systems have been discussed in the

research community, and many lessons from this research have been adopted in commercial database systems. The most well cited academic prototypes include Ariel [73], Starburst [171], SAMOS [68], HiPAC [47] and ODE [98]. A comprehensive survey of the above, including their specific termination, confluence, and determinism properties, is available [119] and serves as an excellent starting point for further research in this area.

4.4.2 Constraint Systems

In the context of enforcement of a large number of constraints, it is plausible that there are avenues for exploiting the interference between constraints. We briefly alluded to the fact that quite often, for example if two states are disjoint, checking for presence in one or the other is adequate to make a conclusion for both. There is in fact a large body of work in the area of analysis of predicates to optimize for precisely these types of situations, where re-writing and re-organizing conditions can lead to significant savings during runtime. In databases systems analysis of predicates is a well understood problem in the context of query optimization [77] and event detection [74]. For a detailed overview of how analysis of predicates and optimal rewriting of predicates is exploited in a commercial DBMS, the reader is directed to a white paper published by SAP/SYBASE [152]. Most importantly, the notion of predicate analysis is inseparably linked with compilers and instruction set execution within modern microprocessors [142]. Although a comprehensive discussion of optimizations through constraint analysis is beyond the scope of this thesis, we believe that advances in compiler technology, which transparently optimizes checking of constraints, will eliminate any need for users to analyze constraints manually and re-organize them for efficient processing during run-time.

We also briefly touched upon the need to analyze properties about the constraint systems developed by the users, that is, to examine the set of constraints from a visual or workflow perspective. Many problems such as satisfiability, state reachability, and workflow dead-ends can perhaps be manifestations of complex policy design errors and should therefore be reported to the policy makers. Much of Chapter 5 is dedicated to similar issues, and we delay further discussion until then.

4.4.3 Summary

In this chapter we examined how to implement constraints derived from our visual constraint modeling language. Much of the research in this area was conducted by researchers in active database systems. Our work presents a bridge from the graphical representation of constraint systems that we proposed to well understood methods of implementing temporal integrity constraints via a space optimized and policy relevant audit trail for individual artifacts. In addition, we analyzed the impact of the business use of such constraints and concluded that, for transactional workloads where individual objects are interacted upon, a database suffers no (or a small constant) penalty over and beyond auditing costs for the enforcement of such constraints.

Chapter 5

Reasoning over Constraint Systems

So far we have discussed constraint specification (Chapter 3) and enforcement (Chapter 4). In this chapter we examine challenges associated with testing constraint systems and answering questions about the possible future(s) of artifacts when subject to a set of DCML constraints (e.g., whether an artifact subject to a constraint systems can ever violate an operational requirement).

Constraints embedded in DCML diagrams are a set of enforceable restrictions specified in constraint temporal logic using past operators only. However, when reasoning about the future, the notion of *enforceability* is replaced by that of *satisfiability*, and thus we need to extend our logic such that statements about the possible future(s) of artifacts can be easily constructed and verified. In this chapter we consider the full past and future fragment of CLTL and include the temporal operators **X** (\circ), **G** (\diamond) and **F** (\square) to represent next time, always, and some time in the future, respectively. Note that these operators are part of classical LTL and thus are also part of CLTL, but since they were not relevant for the specification and enforcement aspect of constraints, we delayed their introduction until now. It shall become evident in this chapter that CLTL restricted to the past is well suited for specifying constraints and restricting the state space for artifacts. However, reasoning about the future is best (most naturally) done using future operators and constructs that allow users to perform forward reasoning in time.

This chapter is organized as follows. In Section 5.1 we demonstrate how the verifica-

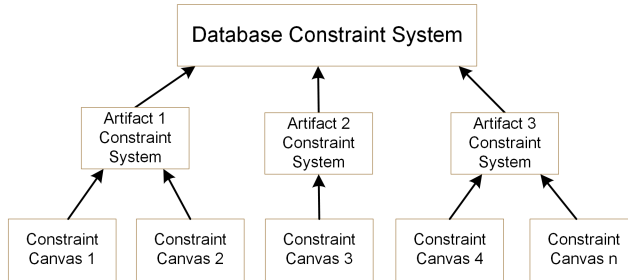


Figure 5.1: Constraints in all DCML canvases are first integrated at the artifact level (unified artifact constraint diagram/system). Although these constraints are specified over the artifact type, they are enforced over all artifacts contained in the database system.

tion of properties pertaining to a set of constraints can be reduced to problem of CLTL satisfiability. We present a method of constructing a CLTL SAT problem, which, when solved, allows us to determine if a user-specified property of the artifact lifecycle can be met under the constraints enacted on the artifact type. While Section 5.1 focuses on verification of properties of a set of constraints associated with a single artifact type, Section 5.2 examines the problem of verifying properties across several interacting artifact types. Finally 5.3 summarizes this chapter and briefly discusses research most closely associated to the work presented in this chapter.

5.1 Model Construction

5.1.1 Terminology

Figure 5.1 summarizes our constraint integration methodology, showing how various DCML constraint canvases are brought together for monitoring and enforcement. Recall that in Chapter 3 we introduced the notion of different policy makers having separate canvases for constraint specification on a single artifact type. Our main reason for doing so was to allow for the distribution and separation of constraint specification responsibilities among different policy makers. However, there is always a single “unified” DCML diagram per artifact type that combines the DCML canvases drawn out by individual policy makers.

With this approach we can reason about the possible behaviour of artifacts at different levels (i.e., at the individual canvas level, the unified constraint system level, and the physical storage/database level). To distinguish reasoning at various levels we define the constraint systems at these levels as follows:

Definition 6. *A canvas constraint system (CCS) is a CLTL formula that is formed by the conjunction of all individual constraints embedded in a single DCML canvas.*

Definition 7. *An artifact constraint system (ACS) is a CLTL formula that is formed by the conjunction of all canvas constraint system(s) acting on an artifact type.*

Recall that each visual construct in DCML introduced in Chapter 3 (Table 3.1 on page 65) embeds one or more restrictions specified in CLTL over the artifact type. Many such restrictions can be embedded in a single DCML canvas created by a single policy maker. If we take the constraints in one such canvas and simply combine them into one logical formula, that is, conjoin each assertion derived from interpreting the graphical notation, we will have achieved a CCS that represents all the constraints in a single canvas. Note that since each artifact must comply with all constraints independently, it must therefore comply with the single combined formula as well.

Similarly, we can collect all the canvases in one unified constraint diagram and this leads us to the notion of an ACS. Note that there is always only one ACS per artifact type, and it represents all the constraints specified by different policy makers on that artifact type. This combined constraint must forever hold true in all individual artifact histories of the type. In other words the ACS captures all the constraints on an artifact type, and each individual artifact's history must comply with the ACS.

This naturally leads us to the definition of a database level constraint system that in some way represents the interaction of all artifact level constraint systems for all user defined artifact types. Note however that we cannot simply combine CLTL constraints specified over disparate objects/domains onto a database via a simple conjunction. Consequently, we defer the discussion of a database level constraint system until Section 5.2. Note that, from this point onwards, we will use the terms CCS and ACS instead of a DCML canvas and a unified DCML diagram for an artifact, since these terms more formally represent the visual notation in CLTL.

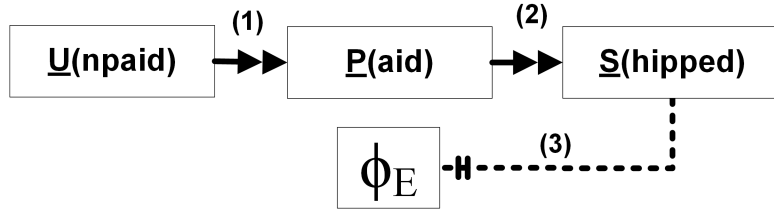


Figure 5.2: A set of restrictions relating and stitching together different states of an artifact (names shorted to U, P and S). An interpretation of this diagram given on a constraint-by-constraint basis could be that (1) if an artifact gets paid it must have been previously unpaid, (2) if an artifact gets shipped it must have previously been paid and (3) a shipped artifact’s lifecycle can never be terminated.

The terminology can be best illustrated through an example. Figure 5.2 depicts a DCML canvas where the InvoiceProcessing artifact (as defined in Section 4.1.2) must go through a restricted set of stages during its lifecycle. To convert this canvas into CLTL constraints, we can once again rely on the notation-to-CLTL mapping provided in Table 3.1 on page 65. In our example we use the short-hand notation of the first letter of the state to denote the function associated with that state applied on the artifact attributes. The resulting constraints derived from each of the constraints labelled 1 through 3 in Figure 5.2 are as follows:

Constraint 1

$$P \wedge \neg \bullet P \Rightarrow \bullet U$$

“If the artifact is in state P and it was previously not in state P, then it must have previously been in state U.”

Constraint 2

$$S \wedge \neg \bullet S \Rightarrow \bullet P$$

“If the artifact is in state S and it was previously not in state S, then it must have previously been in state P.”

Constraint 3

$$\blacklozenge S \Rightarrow \neg\phi_E$$

“If the artifact was ever in state S , then it should not be in state ϕ_E .”

Thus by Definition 6, the CCS for Figure 5.2 is the following single CLTL formula that has to forever hold true in the history of individual artifacts of the specified type:

$$CCS_1(A) := [P \wedge \neg \bullet P \Rightarrow \bullet U] \wedge [(S \wedge \neg S) \Rightarrow \bullet P] \wedge [\blacklozenge S \Rightarrow \neg\phi_E] \quad (5.1)$$

If there are k different CCSs acting on an artifact, then, by Definition 7, the ACS can be defined as follows:

$$ACS := \bigwedge_{i=1}^k CCS_i$$

The reader should observe that an ACS is a conjunction of all the if-then conditions that are created by different policy makers. Each of these if-then conditions is equally enforceable in a compliant history of an individual artifact. Thus, an ACS reduces all the constraints embedded in various DCML canvases acting on an artifact to a single CLTL formula.

5.1.2 The What and Why of Verification

The question of *why* verification really matters was comprehensively addressed in Chapter 1, where we pointed out that one of the major reasons why policy management in database systems is very expensive is that the unintended consequences of various interacting constraints are hard to isolate. Not only can constraints interfere with each other, but they can also interfere with established business objectives and practices pertaining to various artifacts. For example, an incorrectly implemented constraint may cause legal (required) paths in an artifact lifecycle to be un-traversable. More importantly, *proving* (to savvy auditors or to internal managers responsible for specific processes) that unintended consequences of a set of constraints can never occur is extremely difficult when constraints

are programmed in an ad-hoc fashion. To this end, the ability to verify and prove that “something bad” cannot happen is vitally important.

Therefore we need to be able to define *what* this “something bad” is before we can verify that it never happens. We have already demonstrated that CLTL restricted to past operators is an expressive logic for specifying enforceable constraints over artifact histories. However, we now claim that CLTL is sufficiently expressive to describe invariants about artifacts that may warrant verification. Consider our running example of the InvoiceProcessing artifact and a situation where a policy maker wants to verify if there can ever (in the future) exist a “shipped invoice that is subsequently marked as unpaid”, s/he could write that check condition as an LTL assertion $\mathbf{F}(\bullet S \wedge \neg U)$. Similarly the concerned individual can check formulas that specify complex paths in the workflow. For example, if s/he wanted to determine whether it is possible for an artifact to comply with (traverse) the transition from state S_1 to S_3 after being in state S_2 , then the formula $\mathbf{F}(S_3 \wedge \bullet(S_1 \wedge \blacklozenge S_2))$ could be used to represent that particular sequence of state changes.. Thus, it should be apparent that linear temporal logic can also be used to describe invariants, check-conditions, or some prescribed behaviour of artifacts that may warrant verification.

Construction of a CLTL SAT problem

Having written formulas that specify properties (or invariants) about artifacts, then the next natural step is to ascertain whether these formulas can (ever) be true or not. We can phrase this problem as follows: Given a set of constraints applicable on an artifact type (i.e., an artifact constraint system) and a property that warrants verification, can there ever exist an artifact history, that (a) complies with the artifact constraint system and (b) in which the property is true? In other words can the ACS and the property both be satisfied at the same time?

Although this phrasing of the problem reduces the notion of verification of a user-specified property about an artifact to CLTL satisfiability, there are a few more steps left in completing the construction of the LTL satisfiability problem before it can be fed into a CLTL solver for meaningful results. Foremost, note that the ACS does not capture the implicit constraints introduced to manage artifact lifecycle instantiation and termination.

Recall Section 3.4.4 starting on page 60 in which two implicit constraints regarding termination of lifecycles were introduced for every artifact. To simplify our construction we will denote these lifecycle termination constraints as C_ϕ :

$$C_\phi := (\bullet\phi \Rightarrow \phi) \wedge (\phi \Rightarrow \bigwedge_{a_i \in A} \bullet a_i = a_i) \quad (5.2)$$

We remind the reader that the restrictions embedded in C_ϕ were placed to ensure that (i) once the implicitly introduced lifecycle termination marker (ϕ) is set to true, it forever remains true and (ii) the attribute values of the artifact never change from that point onwards. These restrictions were placed to ensure that an object cannot change once it is in the “dead state.”

Finally, recall the instantiation constraint $\phi_S(A)$ introduced in Section 3.5.2 starting on page 67, and note that it was used to represent an instantiation function that is to be interpreted only for $t=0$. With this last component in hand we can completely specify our CLTL SAT problem as follows:

$$\phi_S(A) \wedge \mathbf{G}[ACS \wedge C_\phi] \wedge \mathbf{F}(CheckCondition) \quad (5.3)$$

Note that this CLTL formal is intuitive to understand and requires the solver to find a history/run in which three conditions are satisfied:

- $\phi_S(A)$: The history must satisfy the artifact instantiation constraint.
- $\mathbf{G}[ACS \wedge C_\phi]$: Forever (or **G**lobally), the history must comply with the constraints in the ACS and the termination constraints.
- $\mathbf{F}(CheckCondition)$: Subsequent to satisfying the instantiation constraint, sometime in the **F**uture (or eventually), the *CheckCondition* must become true.

In other words we are requiring a SAT solver to traverse all possible legal artifact lifecycles and determine if (eventually) the specified condition/property can hold true for an artifact subject to a given ACS. If the above formula is satisfiable, then the *CheckCondition* can be met and the “something bad” that the designer of the *CheckCondition* wanted to avoid can happen.

Example

Consider our running example of the InvoiceProcessing Artifact and the check condition introduced earlier: “can a previously shipped invoice be marked as unpaid.” This condition was formulated as $\mathbf{F}(\bullet S \wedge \neg U)$. The ACS for our example was given in Formula 5.1 and C_ϕ was described in Formula 5.2. Since the constraint diagram did not restrict how InvoiceProcessing artifacts can be created/instantiated $\phi_S(A)$ is considered to be *true*. The resulting SAT problem is as follows:

$$\begin{aligned}
& true \\
& \wedge \\
& \mathbf{G}[(P \wedge \neg \bullet P \Rightarrow \bullet U] \wedge [(S \wedge \neg S) \Rightarrow \bullet P] \wedge [\blacklozenge S \Rightarrow \neg \phi_E] \wedge [\bullet \phi \Rightarrow \phi] \wedge [\phi \Rightarrow \bigwedge_{a_i \in A} \bullet a_i = a_i]] \\
& \wedge \\
& \mathbf{F}[\bullet S \wedge \neg U]
\end{aligned} \tag{5.4}$$

Note that the above formula is a simple substitution of the various parts in formula 5.3. For visual clarity we have broken the formula into three different lines. Line one is the initialization, which is unrestricted (i.e., an artifact can take any values to start). Line two is the set of global constraints that the artifact must satisfy. These constraints include not only the operational business rules (contained in the ACS) but also the termination constraints (last two conjuncts in line two). Finally, line three introduces an aspect about the future that the solver must include for while abiding by the rules embedded in the previous

two lines. That is, in the legal (compliant) future subsequent to artifact instantiation, can the specified *CheckCondition* be true?

5.1.3 Bounded Satisfiability and Model Checking

Model checking can informally be described as the process of verifying a given property ψ (often specified in LTL) against a given state system M . A typical model checker encodes the system specification (usually, but not necessarily, in an abstract/meta programming language) and the given property into a satisfiability problem of the form $M \wedge \neg\psi$. In other words, the execution/traversal of the state space of the model of a system and checking if ψ holds in all executions is reduced to the problem of determining if there exists a particular run in which ψ does not hold. A well accepted observation in the model checking community is that execution state spaces can become very large, and it is often the case that violating runs are relatively small. Thus, it may be preferable to check only executions of a bounded length k for violations of the property and gradually increase this length. The process is repeated with increasing values of k (i.e., increasing the execution lengths) until a counter-example is found or k becomes too large and makes the problem difficult to solve given the available computing resources.

Note that there is a significant resemblance between model checking and CLTL satisfiability when model itself is a CLTL constraint. This is evident from formula 5.3, where we can see that the set of constraints enforceable on an artifact describe the *model* and the *CheckCondition* describes the property. That is, the conjunction of the instantiation constraints, ACS, and the termination constraints describe fully the model. The requirement that the *CheckCondition* is eventually met is introduced for a SAT solver to complete the process of verification. Furthermore, if we limit the number of timesteps (sequences of successive attribute assignments to an artifact) that a solver is allowed to consider, we naturally arrive at the notion of *bounded satisfiability* or *k-satisfiability*.

Today, many modern model checkers have direct support for the specification and analysis of CLTL formulas, and thus they can act as *k-satisfiability* checkers for CLTL formulas. If not readily available, support for the analysis of CLTL formulas can be introduced via encodings. For example, Cunha has described an automated method of encoding LTL

formulas as an Alloy model [45], and Rozier and Vardi [133] demonstrate a mapping in the SPIN [80] and NuSMV [35] model checkers.

We believe that there are two main reasons why a model checker is best suited for the task of verifying constraint systems: efficiency and ease of use. Modern model checkers have seen tremendous research interest and provide an extremely efficient environment for the types of constraints that are generated using DCML. They can leverage the strengths of highly engineered SMT solvers that have direct support for reasoning over constraints specified using typical business domains/operators such as integers and strings. However, a much stronger argument in favour of relying on model checkers is that of accessibility. Model checkers are often accompanied by various analysis tools that allow users to explore, optimize, and augment their models in various ways. This is a perhaps a direct result of the popularity of model checking and growth in the community using/supporting it. Thus, reducing our problem to model checking allows us to exploit not only the efficiencies via technical improvements in this field (SMTs, for example), but also to benefit from soft improvements that will eventually help business users understand the results of a problem in a complex policy system.

A note on Zot

In our examples and in the case study presented in this thesis we construct models in the Zot model checker. Zot [124] is a model checking tool geared towards checking the bounded satisfiability for metric CLTL formulas specified over decidable theories and has now been available for more than seven years. Because of its simple LISP-like syntax and the ease of interpreting violating runs (artifact histories or attribute assignments). The main reasons we rely on Zot for constructing and verifying the examples presented in this thesis is that Zot requires very little setup and initialization. Zot can be coupled with several SMT solvers, and users of Zot can almost immediately begin writing assertions in linear temporal logic and verifying them. Consider the following example of a constraint system specified in the CLTL specification language:

$$(counter = 0) \wedge \mathbf{G}(counter = \bullet counter + 1) \wedge \mathbf{F}(counter > 5) \quad (5.5)$$

Intuitively we can see that the constraint system above does the same three things that our SAT construction for verification does. It (a) initializes a variable called *counter* with the value of zero, (b) requires that forever/globally (**G**) the value of the counter is one higher than it was in the previous time step, and (c) that sometime in the future (eventually or **F**) the value is greater than 5. This model is trivially satisfiable in 6 time steps. The Zot model for this CLTL formula presented below is essentially a mirror image of the formula:

```
(asdf:operate 'asdf:load-op 'ae2zot)
(use-package :trio-utils)

(define-tvar 'counter *int*)

(defvar modelconstraints
  (alw
    ([=]
      (-V- counter)
      ([+] (past (-V- counter) 1) 1)
    )
  )
)

(defvar check_condition
  (
    somf
      ([>] (-V- counter) 5)
  )
)

(ae2zot:zot 10
  (
    &&
      ([=] (-V- counter) 0)
      modelconstraints
      check_condition
  )
  :logic :QF_UFLIA
)
```

Listing 5.1: A simple example of a Zot model of a CLTL formula. For simplicity we have split it into three parts, namely modelconstraints, checkcondition and initialization.

The first two lines are required to initialize Zot for CLTL satisfiability checking. The only variable used in this example, counter, is defined in the third line. There is

only one model constraint, and it requires that always (globally) the value of the counter is one higher than it was one timestep in the past (note that arithmetic operators are enclosed in square brackets and $-V$ is used to refer to the value of a variable). Like LISP, the constraints are specified in prefix notation (that is, the operator is followed by its operands). The check condition requires that sometime in the future (somf / \mathbf{F}) the value of the counter becomes greater than 5. Finally, when both the model and check conditions are specified, *Zot* is run for a bound of 10 relying on the logic/theory of quantifier-free linear integer arithmetic (QF_UFLIA) to check whether there exists a run for which the starting condition, the model constraint, and the check condition are all met.

The output of executing *Zot* paired with a solver (*Z3*) is straightforward as well. It is essentially a run that complies with all the constraints. The output for the above model is presented below:

```

---SAT---
LOOPEX = TRUE
LLOOP = 10
----- time 1 -----
COUNTER = 0
----- time 2 -----
COUNTER = 1
----- time 3 -----
COUNTER = 2
----- time 4 -----
COUNTER = 3
----- time 5 -----
COUNTER = 4
----- time 6 -----
COUNTER = 5
----- time 7 -----
COUNTER = 6
----- time 8 -----
COUNTER = 7
----- time 9 -----
COUNTER = 8
----- time 10 -----
  **LOOP**
COUNTER = 9
----- end -----
Evaluation took:
  0.131 seconds of real time
  0.046800 seconds of total run time (0.031200 user, 0.015600 system)
  35.88% CPU

```

```
287,516,718 processor cycles
758,440 bytes consed
```

Listing 5.2: Output of the *Zot* model showing a run in which counter exceeds the value of 5. In case of an unsatisfiable formula *Zot* returns only with UNSAT.

Complete Example in *Zot*

Finally we conclude this section by converting the SAT problem in Formula 5.2 into a *Zot* model. The *Zot* encoding for this model is once again straightforward and is given below:

```
(asdf:operate 'asdf:load-op 'eezot)
(use-package :trio-utils)
;Variables used: PAID, UNPAID, SHIPPED, PHI

(defvar Instantiation_Constraints
  (!! (-P- PHI))
)

(defvar Termination_Constraints
  (&&
    ;Previously PHI implies currently PHI
    (alw ;Always or Globally
      (
        ->
        (past (-P- PHI) 1) ;Previously PHI
        (-P- PHI); Currently PHI
      )
    )

    ;PHI implies all attributes retain previous values
    (alw
      (
        ->
        (-P- PHI)
        (&&
          (|| (&& (-P- PAID) (past (-P- PAID) 1)) (&& (!! (-P- PAID)) (!! (past (-P- PAID) 1))))
          (|| (&& (-P- UNPAID) (past (-P- UNPAID) 1)) (&& (!! (-P- UNPAID)) (!! (past (-P- UNPAID) 1))))
          (|| (&& (-P- SHIPPED) (past (-P- SHIPPED) 1)) (&& (!! (-P- SHIPPED)) (!! (past (-P- SHIPPED) 1))))
        )
      )
    )
  )
)
```

```

(defvar Model_Constraints
  (&&
    (alw
      (
        ->
          (&&
            (-P- PAID)
            (past (!! (-P- PAID)) 1)
          )
          (past (-P- UNPAID) 1)
        )
      )
    )
  (alw
    (
      ->
        (&&
          (-P- SHIPPED)
          (past (!! (-P- SHIPPED)) 1)
        )
        (past (-P- PAID) 1)
      )
    )
  (alw
    (
      ->
        (somp (-P- SHIPPED))
        (!! (-P- PHI))
      )
    )
  )
)

```

```

(defvar check_condition
  (somp
    (
      &&
      (-P- UNPAID)
      (past (-P- SHIPPED) 1)
    )
  )
)

```

```

)
(eezot:zot 10
  (
    &&
    Instantiation_Constraints
    Termination_Constraints
    Model_Constraints
    Check_Condition
  )
)

```

Listing 5.3: The complete Zot model for the example discussed in this section. For simplicity each of the states is considered to be a boolean variable.

Observe that there is very little effort required to encode the entire constraint system as a Zot model, as each connection between two states maps to an individual constraint in the ACS (Model_Constraints). This entire process is made convenient by the ability to separate sub-formulas into different components and the direct availability of temporal and logical operators in the Zot modelling language. A general purpose DCML diagram to Zot model translator can easily be developed using the rules presented in Table 3.1 on page 65.

5.2 Database Level Constraint Analysis

So far our discussion of verification of properties about constraint systems has focused on a single artifact. Figure 5.3 depicts this situation, where multiple policy makers working on different canvases (but still using the same variables) create a set of constraints visually. Since each DCML canvas is associated with the same artifact type, these constraints can be integrated together in an ACS (or equivalently a unified DCML diagram) and then properties formulated about the artifact that warrant verification. Figure 5.4 depicts a significant benefit of this approach of “unit testing” smaller lifecycles (or portions thereof) embedded in various CCSs before integrating them into one large system of constraints.

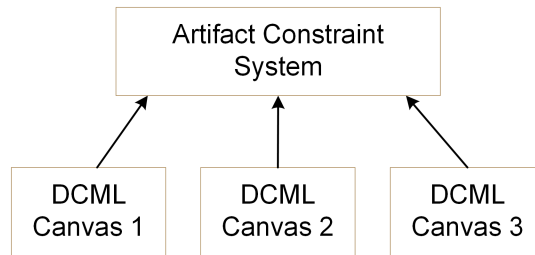


Figure 5.3: Different policy makers can work with the same definition of an artifact type to specify different (sub)constraint diagrams. Because of the way these systems are designed integration of constraints over a single artifact type (conjunction of constraints) poses no challenge.

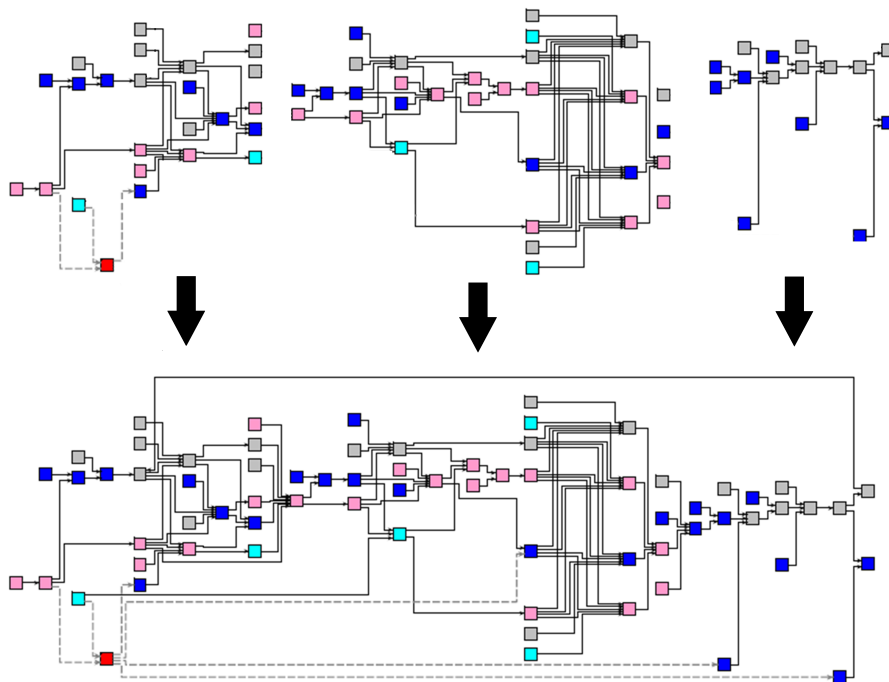


Figure 5.4: Different policy makers can work on individual components of the lifecycles of artifacts. These lifecycles can be unit-tested and eventually be integrated as one unified artifact constraint system.

5.2.1 Defining Conflicts and Errors

Before we proceed to examining issues in database level constraint analysis (i.e., verification of properties across several different artifact types), it is important that we define precisely the term *conflict* as it pertains to an artifact’s lifecycle and the notion of check conditions specified as properties in CLTL.

Let us consider a situation where the *CheckCondition* created by a user represents a “desirable property” of an artifact. An example of a desirable property for our Invoice-Processing artifact could be as follows: it should be possible for an invoice to get paid ($\mathbf{F}(\textit{paid})$). In other words, the constraints imposed by policy makers should allow for invoices to get paid. If however this *CheckCondition* is unsatisfiable when tested against the constraint model, then the artifact lifecycle is over-constrained. Next consider the opposite situation, where the *CheckCondition* represents an “undesirable property,” such as earlier example of an invoice that is marked as shipped and then subsequently marked as unpaid. If this particular *CheckCondition* is satisfiable then we essentially have a situation where the artifact lifecycle is under-constrained. Note that typically the notion of conflicts (i.e., conflicting constraints enacted by error) most directly relates to the over-constrained situation.

Although in both the above scenarios we posed a CLTL satisfiability problem, the result (satisfiability or unsatisfiability) can lead us to very different conclusions about the situation. This is because the the *CheckCondition* represents a condition similar to liveness in the first situation and safety in the second. However, from this point onwards we will use the terms policy level error and policy level conflict interchangeably to imply a situation where one of the outcomes (satisfiable or unsatisfiable) on a *CheckCondition* is interpreted as a negative result in terms of accomplishing a business objective.

5.2.2 Cross-artifact Reasoning

So far we have examined constraints specified on a single artifact and seen that canvas constraint systems on the same artifact can be very easily brought together in an artifact constraint system. However, when integrating artifact level constraint systems across mul-

tuple base tables, several issues arise which make a direct conjunction of different artifact constraint systems difficult. To provide a concrete example of the intricacies involved in the verification of multi-artifact properties across the contents of the database, let us consider three tables (two of which were previously introduced) and two new artifact definitions that share the new common table:

CUSTOMER: (int **CST_ID**, varchar NAME, varchar ADDRESS, . . . , varchar ADDRESS_COUNTRY, varchar CST_TYPE)
 INVOICE: (int **INV_ID**, datetime LASTUPDATE, int CST_ID, float AMOUNT, bool SHIPPED, bool PAID, bool APPROVED, . . . , varchar NOTES)
 SHIPMENT: (int **SHIP_REF**, datetime LASTUPDATE, int CST_ID, datetime SHIPMETHOD, . . . , varchar NOTES)

To keep this example to the point, we shall omit drawing out individual states/diagrams and only consider constraints in CLTL. Let us consider the following two artifacts and their respective constraint systems. For further simplicity, each ACS has only one constraint in it.

OrderShipment Artifact: SELECT * FROM INVOICE NATURAL JOIN SHIPMENT

Constraint 1: If an invoice is worth less than \$1000, do not use the “overnight” shipping method.

Constraint 1 in CLTL : $\bullet (amount \leq 1000 \wedge \neg shipped) \Rightarrow \neg (shipped \wedge shipmethod = \text{“overnight”})$

CustomerShipment Artifact: SELECT * FROM CUSTOMER NATURAL JOIN SHIPMENT

Constraint 2: Purchases for business customers must be shipped overnight.

Constraint 2 in CLTL : $(cst_type = \text{“Business”} \wedge shipped) \Rightarrow (shipmethod = \text{“overnight”})$

This scenario is extremely simple and can be summarized as follows. A policy maker from the Sales department has established a policy (constraint 1) on the OrderShipment artifact that requires that low valued invoices, those valued less than \$1000, must not be shipped using the overnight shipping method. At the same time the customer service department has a policy (constraint 2) on the CustomerShipment artifact that requires that in order to receive high priority all business customers' purchases must *only* be shipped using the fastest method possible, which is overnight shipping. It is important to keep in mind that although this is an extremely simple and concocted example, such a system of requirements could emerge quite commonly in business situations. For example, either constraint 1 or constraint 2 could have been present and a new policy directive may now require that the other be implemented as well.

Error Detection and Virtual Artifacts

Observe that both these constraints are independently correct and enforceable. However, the end result is that if a business customer purchases items of value less than \$1,000, there is no way for that order to be shipped. This implication will not be easily apparent, since the two constraints are enforced over different combinations of tables (i.e., different artifact types). A customer can have many shipment records as long as these records in their lifecycle meet constraint 2, and, independently, orders can have associated shipment records as long as these records in their own lifecycle meet constraint 1. However, it is only when we examine all three entities (a shipping record for a given customer's invoice) at the same time and formulate a proper argument (in the form of a temporal property/assertion) about them that we can reach meaningful conclusions about the interplay of shipping different valued orders by business customers.

In other words, if we consider the question of whether there exists a way to ship a low valued order placed by a business customer, across all three tables, the obvious unsatisfiability result will follow. It is important to note that when examining only two tables at a time we leave the attributes of the third table as free variables in any SAT problem. Since the two-table at a time approach does not fully model the three-table situation at hand, it is incapable of broader level reasoning regardless of the properties checked against it.

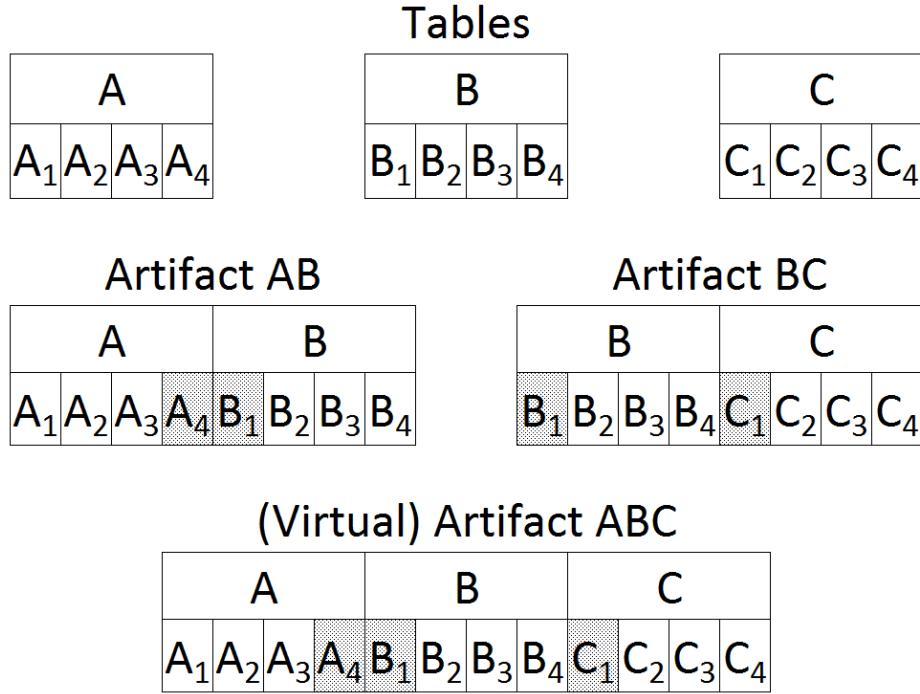


Figure 5.5: A database with 3 tables and three artifacts that join on the shaded variables (A_4 , B_1 and C_1). Reasoning over artifacts AB and BC does not capture reasoning over the broader model represented by the artifact ABC. This is because individually, models over the artifacts AB and BC are agnostic to the attributes not represented in them. The shaded variables represent a join condition, which in CLTL can be expressed as $Join(A, B) := \neg(\bullet\phi_S \vee \mathbf{U}(\phi_E)) \Rightarrow [(\bullet A = A) \wedge (A = B)]$. A join condition between A and B requires that if the artifact was not previously in ϕ_S and not going to ϕ_E in the next time step, then the previous value of A is the present value of A and the present value of A is the present value of B. Thus variables A and B are joined once at the first time step and will perpetually retain the same value in all executions explored by the model checker.

Consider the abstract visual depiction of this example presented in Figure 5.5. Observe that constraints defined on the artifact AB are only applicable on the attributes of Tables A and B, and similarly constraints defined on artifact BC have no direct bearing on the attributes of Table A. From a verification perspective, properties and assertions written against each of these artifacts pose the questions of whether there exists:

1. a pair of elements from table A and B that join on A_4 and B_1 and fail to comply with a given assertion or;
2. a pair of elements from table B and C that join on B_1 and C_1 and fail to comply with a given assertion.

Therefore, in situations where the reasoning is restricted to a given set of tables, the model and property under question do not consider constraints that exist in other potentially related artifacts. To address this issue, the user that is attempting to verify policies has to ask the right question across several tables and essentially force the join condition to be true: That is, for the given constraint model across all three tables, can there exist tuples from tables A, B, and C, that simultaneously join on A_4 , B_1 and C_1 and also fail to comply with a given assertion. By forcing the join, the user has extended the constraint model (view ABC in Figure 5.5) and expanded the vocabulary (list of attributes) on which assertions can be verified. We denote such views that represent artifacts or interactions between artifacts that are not explicitly defined for policy purposes as *virtual artifacts*.

At this point it should be apparent that the quality of the conclusions drawn about the behavior of a database system will be only as good as the quality of the individual policy models and the questions posed against them. Furthermore, properties checked against individual artifact constraint systems may provide significant insight to the individual models, but these properties are not sufficient for reasoning over the entire database. To properly reason across several artifacts, we have to examine all base tables that they refer to and then introduce individual constraints and appropriate joins clauses for each of the related pairs of artifacts.

Complicating the situation even further is the fact that when several artifact constraint systems are integrated together over a database, there is no inherent notion of correctness

derivable from the properties specified on the individual artifact constraint systems. For example, in the given set of specifications that we just discussed, there was no assertion mandating that all orders should be “shippable,” and it may very well be the case that business users are required to have a minimum purchase of \$1,000 or higher for their order to be processed. Thus, it is impossible to know whether what seems like an obvious problem in the integrated model (virtual artifact) is actually a cause for concern or just a well thought out consequence. As a result we have to rely on the users/testers to provide an encompassing definition of what should happen (definition of a broader/virtual artifact as depicted in Figure 5.5 and desirable properties thereof) in the combined set of constraints acting on a database.

5.2.3 Making Life Easier for the DBA

The main challenge in the integration of various artifact level constraint system is that of developing global models (virtual artifacts) within a database, integrating individual constraints applicable on these models, and then specifying properties to validate over these sets of models and constraints. This problem shares several aspects of similarity with that of unit-testing/system-testing in software engineering and compositional model checking [37]. In all of the above cases, users accomplishing these tasks require a significant understanding of the system that is being modeled (constrained). Here the word *user* refers to a tester or an auditor of the database level constraints, and the expectations from him/her are no different than a typical software tester. S/he must have a thorough understanding of the system’s expected behaviour and be able to articulate possible avenues of failure in the appropriate testing language (CLTL).

Although we claimed earlier that many policy makers can diagrammatically formulate artifact lifecycles on their own, we now have to acknowledge that there will be far fewer power users who will be assigned the role to verify global properties about the interplay of artifact constraints systems. Additionally, these power users will have to report potential problems back to the original policy makers and play an active role in mediating conflicts. The most likely candidates for this job will be database administrators who need to be aware of the intricacies involved in temporal reasoning and possess knowledge regarding

the business intent of the entities and relationships present in the database.

Developing Database Level Constraint Models

Developing test cases against complex systems consisting of several components is clearly not an easy task, and it may even be considered an art. Without trivializing the problem we can see that there are many immediate directions that a DBA can follow in verifying properties about the interacting lifecycles of various artifacts.

In most situations, the source of contention will invariably be the data that is shared among several artifacts. Clearly if a relational schema has clusters of unrelated tables, then it stands to reason that the constraints in one isolated set of tables should not interfere with another. As the example presented in Figure 5.5 showed, joins across tables in the artifacts definition typically leads to the sharing of underlying data across many different artifact definitions. In the case of Figure 5.5, a broader model (virtual artifact ABC) had to be defined so that it encompasses both artifacts and is capable of expressing (unifying) constraints on both artifacts and simultaneously reasoning over them.

Unfortunately, there is no formal algorithm that can generate global models that will be of interest to a given tester. Carefully sifting through combinations of smaller systems that can be progressively integrated is a task left to the judgement of the DBA.

Development and Verification of Complex Properties

Let us say that a DBA/tester is able to identify a situation like the one presented in Figure 5.5 and that a model that encompasses multiple artifact constraint systems is readily available. The discovery of conflicts still requires that appropriate properties be formulated against this model. In many situations these properties can be developed by looking at system specifications. Consider, in light of our running example, the emergence of an overarching requirement of the model (somewhat similar to a liveness property), requiring that, for all types of customers and amounts, an order should be able to eventually reach the shipped state. Since this requirement cannot be met in our example, we have a conflict.

Note that this is the first time we have introduced a quantifier (universal quantification over the domain of customer types and amounts) in our scope of reasoning. In the next chapter we go into greater detail about the implications of having a quantifier-free constraint specification language and our reasoning for choosing the particular fragment that we have. For now we only point out that these properties that a tester/auditor may want to verify need not be in the same quantifier-free fragment of LTL that we proposed for constraint diagrams. In fact, the tester can rely on as much expressivity as the situation requires. For our constraint system specified over finite domains, we can repeatedly ask a solver for every combination (cross product) of types of customers and amounts in the invoice to identify runs where the order is shipped. Although this is not a very efficient approach, we will invariably encounter the situation in which a business customer orders goods of less than \$1,000, leading to an unsatisfiability/unreachability condition for the shipped state. In practice, the knowledge about the underlying domains and relevant theories help us avoid exhaustive enumeration. Knowledge about the theory of linear integer arithmetic can lead us to conclude (as it would for a programmer testing a piece of code) that testing only boundary conditions (\$999, \$1000 and \$1001) is sufficient to make sound conclusions.

The knowledge about domains is of significance during this stage. We have already assumed that the model tester (DBA) is a highly capable person. Thus, it stands to reason that s/he should be free to introduce additional domain knowledge to augment or further restrict the model if it leads to better (faster or more efficient) reasoning. For example, instead of enumerating over all possible strings, a DBA can make his/her life easier and restrict enumeration of shipping methods and customer types over a short list of possibilities defined either in the constraint system itself or available from within the database. By giving the tester utmost flexibility (much more than the designers of the canvas constraint systems had), we can certainly provide a bigger and better arsenal of tools for him/her to exploit. However the flip side of this is that it expands the scope of responsibilities, making the job of the DBA even harder when there are no set criteria for correctness.

5.3 Summary and Related Work

In the previous chapter we looked at how a given a set of constraints in DCML can be enforced. In addition, we examined several issues pertaining to the feasibility of enforcement. In contrast, this chapter made the case that developing a “correct” system of constraints at either the artifact level or the database level is an extremely challenging task. Although empowering business users to create constraint diagrams on their own is a noble idea and reduces the burden on a single authority to monitor/manage a database, it is certainly not without its perils. We observed that even though the developed constraints will always be enforceable, the end result of integrating many constraints and enforcing them can be the introduction of latent problems that may need to be identified before implementation. Thus, we distinguished the notions of enforcement and correctness and brought forward the idea of testing a system of constraints.

Underneath the infrastructure for reasoning that we presented is a very simple observation: A database in its purest form is a set of unrestricted tuples. It is essentially a “universal model,” as described by Rozier and Vardi [133], where tuples can “move” or switch between any arbitrary values as time progresses. It is only when we start assigning meaning to the different attributes, give them domains, give interpretations to joins among these tuples, add constraints (primary key, foreign key, domain constraints, assertions, checks etc.) that we start to build a scenario that restricts the universal model to suit a particular situation. Our approach augments a database with additional constraints specified in CLTL that are applicable over the lifecycle of a single unique combination of such tuples.

A consequence of our choice to use CLTL to specify constraint systems is that the problem of verification of such systems is essentially reduced to CLTL satisfiability and model checking. Several surveys and text books on model checking have been published, and it would be a digression to discuss them. In addition to the work done with Rozier, Vardi has on two different occasions pointed out the similarities between the research conducted in the areas of satisfiability and model checking and the connections therein to database systems. He demonstrated that CSP/SAT problems can be reduced to several problems in query analysis and vice-versa [166]. Later he noted that many database-centric problems, such

as serializability of transactions, can be framed as model checking problems [167]. Several other authors have also presented work modeling the behaviour of concurrently running transactions and testing their behaviour [9, 53]. Note that our foremost goals were that of interpreting workflows, constraint specification, and enforcement in database systems. However the additional infrastructure of verification through model checking comes as an additional benefit that results from our choice of language to develop constraint systems.

The idea of using model checking (and reasoning over state systems in general) to abstractly identify inconsistencies in business processes has also been explored several times [121, 158, 56, 12]. Our work is similar and relies on establishing a mapping between workflows and enforceable constraint systems while simultaneously enabling verification over these constraint systems. Most importantly, our approach distinguishes itself by being concrete, and attempts to reason at the level of individual database level rows instead of abstractly defined business objects.

Chapter 6

Usability, Limitations and Extensions

In this chapter we revisit the goals we set in the introduction and examine several unanswered questions about the *usability*, *limitations*, and *extensibility* of our proposed technique for enforcement and static reasoning with constraints. We demonstrate that policy makers can extend DCML to suit their particular needs and introduce a broad range of constraints to transfer their business level workflows over to a database. Recall that in Chapter 3 we presented a limited number of constraints with different notations (arrow ends) to define DCML (Table 3.1 on page 65). In this chapter we will make the case for an extended situation-specific DCML toolkit that can be customized for users attempting to enforce various types of policies in relational database systems.

There is a fine balance between *extensibility* and *usability* that is often determined by the *limitations* of a complete system. In our case the three topics are closely related by the logical formulation of our solution. This chapter takes a closer look at this relationship and highlights some important observations. As was the case in the previous chapter, we take a broad meaning of the word “user” and consider the notion of usability from both a policy creation/visualization and policy testing/management perspective.

This chapter is divided into four sections. In Section 6.1 we take a closer look at the key determinants of expressivity of constraints in our proposed framework. We examine the impact of the expressivity of the underlying query language used to define artifact types (SQL) and the expressivity of the language used to describe constraints over artifacts

(CLTL). In addition we discuss how the interplay between these two imposes limitations and impacts the overall usability of an LTL-based visual constraint modeling language such as DCML. In Section 6.2 we consider three possible extensions of DCML. These extensions add a significant amount of expressivity in the types of constraints that can be enforced on artifacts while maintaining the overall usability of DCML. In Section 6.3 we take a more formal look at workflow languages and demonstrate that DCML (and, more generally, LTL) is sufficiently expressive to capture the control flow characteristics of most modern workflow languages. We rely on the work done by van Der Aalst et al. [163] to characterize the features present in workflow languages as “patterns.” It is shown that most workflow patterns (specifically those relevant to a single class of objects, i.e., a single artifact type) can be mirrored in an LTL- based visual constraint modeling language such as DCML. Finally, Section 6.4 summarizes and concludes this chapter.

6.1 On Usability and Complexity

The reader familiar with model checking and the relationship between SQL queries and first order logic will have observed that there are three major determinants of complexity that impact how efficiently policies can be enforced and if they can be statically (and meaningfully) verified. They are as follows:

- the expressiveness of the view (query) that defines an artifact type,
- the complexity of the artifact constraint system, and
- the complexity of the check condition that requires verification.

Chapter 4 examined issues pertaining to the complexity of the artifact definition and the impact of the types/frequency/coverage of updates issued against the database on efficiency in constraint enforcement. It analyzed the compromise between storage space and computational efficiency that the database administrators may have to face. The problems related to the efficient monitoring of a database for LTL constraint enforcement are well known in the database community in the context of triggers and materialized view

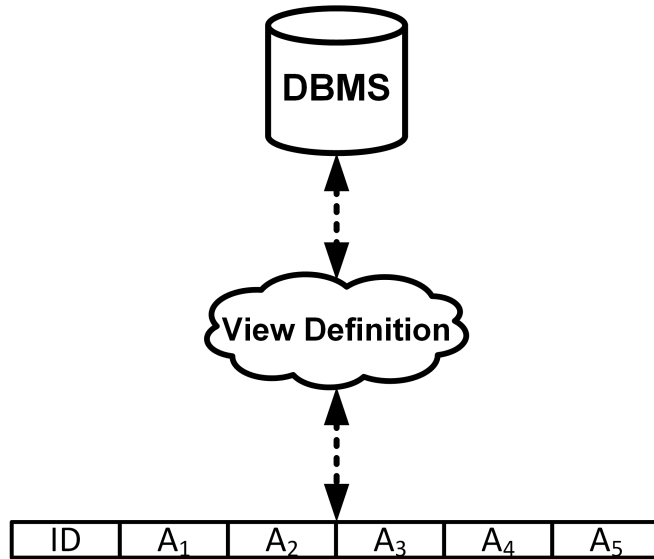


Figure 6.1: Restricting the state space of n-tuples defined over a database requires that we analyze the implications of how these tuples are derived. For example, if the query defining the view is unsatisfiable the entire constraint system is essentially meaningless. For arbitrary unrestricted queries, determining satisfiability is well known to be undecidable. Similarly, when sufficiently expressive views are utilized, reasoning over constraints that cannot be mapped back onto the physical contents of the database is severely impeded. Even though such constraints may be enforceable, they provide limited knowledge about the possible behaviour contents of a database.

maintenance. Chapter 5 addressed the problem of verification of properties over artifacts that are subject to a set of complex constraints. However, there remain several questions regarding the expressiveness of the artifact type (view) definition language (i.e., SQL), the expressiveness of the constraint specification language (i.e., CLTL), and how they impact our ability to gain meaningful insight about the constraint systems that have yet to be addressed.

Figure 6.1 captures many of the issues that emerge because of the mapping from the artifact as a uniquely identifiable n-tuple to the contents of the database over which it is defined. The primary purpose of having this layer of indirection is to provide each policy

maker with a clean snapshot of a specific class of business objects (artifact type) scattered in different tables. The benefits of doing so (primarily simplification and distribution of roles and responsibilities) have been repeatedly stressed in this thesis. However, we need to address the fact that ill-defined views (artifact types) can lead to significant difficulties (discussed shortly hereafter) in verifying properties about ill defined artifacts.

Most of the examples of artifact types presented in this thesis have been simple, everyday businesses objects, such as invoices, sales records, etc. The contents of such artifacts are directly stored in the database and do not require any operations (manipulations or modifications) to be captured in a view. The reader familiar with first order logic and relational algebra will recognize that such views are defined using conjunctive SPJ (Select-Project-Join) queries and are also referred to in the literature as existential queries. Their key advantage is that they make the connection between existence/counter-example generation much easier to be understood. This is due to the fact that, when considering SPJ-based artifact types, a direct connection to constituent rows is available to explain the history of artifacts.

It is generally accepted that many “typical business records” can be captured using SPJ views. In this thesis we will not delve into why SPJ views are technically much easier to reason with than more expressive views. The reader is however directed to prior classical work that have done substantial analysis on the satisfiability and decidability of classes of queries under varying assumptions (e.g., [94]). However, we argue that technical limitations aside, there are major non-technical limitations imposed by the use of complex views to define artifacts, specify constraints over them and interpret the meaning of those constraints for policy makers. We will demonstrate that, in the context of policy verification, restricting artifact definitions to conjunctive queries (or the related notion of updatable views) provides the perfect balance of usability and complexity for the class of business level constraints we have targeted in our work.

6.1.1 Meaningful Constraints over Complex Views

To illustrate the “meaningfulness” of constraints over complex views, let us consider Figure 6.1 and an artifact that contains the aggregate function SUM() over an integer attribute

in a database table. One can argue that we have gone beyond the capabilities of first-order logic by using aggregate functions, but it is important to note that we have only done so in defining the artifact. Reasoning about the possible values this sum can take is still practical.

It is important to recognize that a constraint on the sum of a set of values does not constrain any of the individual values of items in a set. For example, if the sum of a set of values is required to be less than zero, the set itself is not restricted in any way and can contain any number of positive and negative values, provided they collectively add up to an amount less than zero. For the purposes of verification, constraints specified over such derived attributes will be meaningless and have no bearing on how the actual data values of a single artifact behave. Nonetheless, even though we lose the strong connection between a database state space and the artifact state space when going beyond conjunctive/SPJ queries we can (a) still enforce constraints on complex artifacts and (b) perform some reasoning at the level of the artifact.

6.1.2 Complex Functions and Constraints over Sets

However there is a big caveat here: through the use of non-trivial functions, we also lose our ability to reason across constraint systems. For instance, consider the case where one policy maker in one constraint system requires that all individual values for a given attribute in a table will forever be less than zero and another policy maker in a different constraint system (enacting constraints on a different but related artifact type) requires that the aggregate of the same values must forever be greater than zero. Since an off-the-shelf SAT solver / model checker has no knowledge about the theory of sums over unbounded sets, it is impossible to correctly verify whether a set of values can satisfy both these constraints. In other words, in the current framework we cannot integrate these two constraint systems without bounding the set size and encoding the equivalent finite problem in a SAT solver.

Verification in such situations can thus only be accomplished when we have available proper frameworks (theories) for reasoning with the appropriate functions. This, of course, also includes the infrastructure to model temporal properties over arbitrary sets of variables and not just a single tuple. In addition to summation, user-programmed functions pose

the same problem. If the model checker is augmented to reason over such functions, then clearly the query language to define artifacts can be also extended. That will, in turn, allow us to formulate properties over systems that interact with each other through constraints on the underlying data. However, without this additional infrastructure, complex set-oriented functions in the constraint systems on their own will have limited usability (i.e., only at the level of an artifact) and in general be unable to provide guarantees to testers of a system attempting to bring together different constraint systems.

6.1.3 A Case for Reduced Complexity

A much stronger argument in favour of avoiding arbitrary complexity lies in the examination of how typical business workflows and process models are organized and why the proposed class of logic/constraints is ideal to support them. When specifying CLTL constraints over SPJ views, we have (i) restricted ourselves to a single existential quantifier in first order logic (i.e., verification questions that ask whether there can exist an illegal sequence of history entries for a single object) and (ii) only allowed the use of temporal operators over atomic propositions in DCML.

Note that with regards to point (i), the lack of the universal quantifier restricts us to reason over the existence of a single artifact (or virtual artifact) in isolation. This is convenient for users because most policy situations emerging from workflows consider only the lifecycle of single class of objects. In other words, since all instances of a workflow follow the same set of rules, finding one instance that violates those rules is sufficient to discover an error.

Point (ii) above relates to the fact that arbitrary use of temporal and arithmetic operators within propositions is avoided in our constraint system construction. When defining a particular state, the object is “frozen” to its “current” values. For example, using DCML a constraint such as $\bullet(x < y)$ can be generated because at a particular point in time we can compare attributes of an artifact using the $<$ operator and refer back to that state condition being met as part of a state transition. That is, a policy designer can define a state where the *current* value of x is less than the *current* value of y and refer back to this

state implicitly when embedding restrictions in LTL. However, given the way we have defined DCML, a state itself cannot be defined by arbitrarily using temporal and arithmetic operators, e.g., $(\bullet x < \blacklozenge y)$ cannot be defined. Thus, a state in which the previous value of x was less than some prior value of y cannot be modeled by the visual construction notation. Consequently, one can argue that from a technical standpoint some expressivity and flexibility in reasoning (well within the capabilities of a model checker as exhibited by Listing 5.2 on page 118) is perhaps unnecessarily lost.

We believe that this “lost expressivity” provides enormous usability for the users of such a policy management framework. The biggest gain in usability comes from the fact that canvas constraint systems become very closely related (visually) to path-oriented workflows rather than exotic automata. “States” directly relate to stages in a business process, and these stages are themselves identifiable based on the properties of a business object. The most appropriate real-world analogy to this situation is that of an assembly line, where a worker familiar with the process can look at a widget (and the widget alone) at any time and determine what stage of the assembly process it is in. This restriction precludes potentially some exotic class of business processes that have loops in them and the number of times an object has been through a specific loop is of significance for future decision making. Our work, however, focuses on the more common situation: by looking at an object’s attributes alone, a policy designer is able to identify what stage of the workflow it is in. Consequently by not providing the additional expressivity, we ensure that the mapping from business level workflow states to artifact states is straightforward and easy to accomplish for the most common class of workflows. In the next section we discuss how the overall expressiveness of systems of DCML constraints can be increased on a situation-specific basis.

6.2 Enhanced Constraint Systems

6.2.1 Extending the DCML Toolkit

The previous sections should lead the reader to appreciate that there are indeed many issues that lie at the intersection of usability and technical limitations in our proposal. By

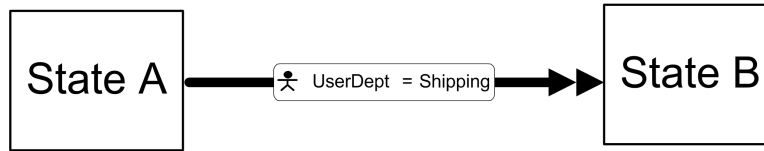


Figure 6.2: An example of a workflow constraint that mandates, in addition to the exit restriction, that the user updating the object (causing it to go from State A to State B) must belong to specified user group.

the end of this chapter, we will be able to summarize them and provide a clearer description of the scope of features that can be supported and the specific limitations they introduce. To illustrate the broader point and at the same time strengthen our proposed model, we now present what we consider “usable extensions” of DCML, which provide a significantly increased array of policy enforcing/modelling tools accompanied by limited compromise in usability. We focus on three such extensions, and in each case we discuss the benefits and consequences of providing the additional facility to policy makers.

6.2.2 Metadata and Temporal Access Control

So far we have focused on enacting and reasoning over policies on artifacts whose contents are directly available in the underlying tables of a database. However additional metadata regarding individual objects in a database may be useful for policy making. Most database systems can store, alongside the history of an artifact, additional details such as the user name, user group, role, and purpose of the transaction that initiated the change in an artifact. The instrumentation to store this metadata alongside artifacts can easily be embedded in the underlying triggers that maintain the artifact history. The definition of the relevant view/artifact/object history in such situations can be visualized as follows:

$$ObjectHistory = (ID, timestamp, A_1, A_2, \dots, A_N, user, usergroup, txntype, application, \dots, purpose)$$

The additional attributes *user*, *usergroup*, etc. represent metadata from the event acting on the object. Transactional metadata is often 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. Consequently, many classes of business policies are specifically related to not only the contents of an artifact before and after an update, but also to the user or the application committing that change (Figure 6.2). Other examples of such policies include specifications that require “expense claims of over \$50 in value must be approved by the user named Wendy” or “if an invoice is being moved from the *not shipped* state to the *shipped* state, then the change must have been initiated by a user in the logistics user group.”

Process-oriented temporal access control restrictions are generally very difficult to model in languages that are not state oriented. However, if artifacts’ histories were to encapsulate access control events as part of their definitions, then we can easily define states based on conditions associated with this additional metadata. Consequently, we could integrate process-centric, temporal access control models with business process models, and they could be validated/tested within the same constraint system.

Furthermore, within the framework of policy/artifact constraint systems, we can also accommodate a mix of temporal and non-temporal access control policy models. Applications of such systems include situations where different paths are taken in the lifecycle of an object based on the privileges of certain individual users. Once an object takes a particular path at such a juncture, it need no longer be restricted by the initial access control constraint. For example, this could be applied when a critical decision needs to be taken by a user who has an elevated access control privilege specific to that process (such as a supervisor), but subsequent to this decision the process need not be restricted in terms of access control. Similarly, access control event chaining can be accomplished in a manner similar to generalization of state-oriented constraints, allowing us to specify complex conditional temporal access control 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.”

Limitations

Implementing temporal access constraints requires very little effort in addition to augmenting definitions of artifacts to store historical metadata. Once the triggers that maintain histories of artifacts are modified, additional metadata can be treated as regular attributes to define states and enact policies. Thus, access control restrictions are rephrased as data integrity restrictions constraining how the artifact can evolve.

However, there is a major caveat for policy makers attempting to reason over such constraint systems. This business process level reasoning does not eliminate the underlying access control layers present in the database server and the operating system. That is, if an independent access control mechanism within the database engine explicitly rejects a modification, then whether a constraint system allows it or not becomes irrelevant. The temporal access control restrictions within a constraint system essentially augment the access control rules already present, and a transaction is required to pass both levels of constraints. The above fact implies that the job of the DBA is made more challenging when verifying properties about constraint systems involving access control. More specifically, the DBA must also incorporate the behaviour of the database and operating system level access control systems.

Fortunately, in most modern database systems the access control layer is relatively simple to model. It primarily consists of GRANT/DENY statements for INSERT, UPDATE, and DELETE operations on tables for specific users and group. The implications of each of these conditions has to be integrated with the constraint systems. For example, if a particular user does not have permission to modify rows from a particular table, this restriction must be appropriately transferred as a CLTL restriction on the appropriate views involving the table in question. In case of INSERT and DELETE permissions, the corresponding restrictions will be modelled by restricting $\phi_S(A)$ and/or never entering ϕ_E , if the transaction is initiated by a certain user/usergroup. In case of UPDATE, we will require a global assertion (**G**) mandating that a specific user is unable to modify the artifact.

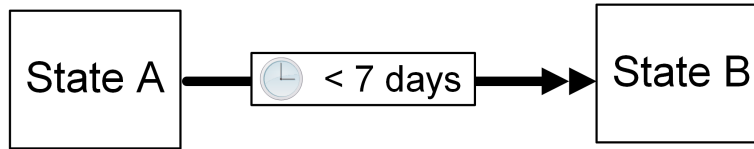


Figure 6.3: An example of a workflow level constraint to enforce an enhanced exit restriction that requires that the change taking an artifact from state A to state B must take place within 7 days.

v

Read access to the database

It is also important to remind the reader that our focus is restricted to maintaining integrity only; privacy restrictions are not supported. This is in contrast to general-purpose access control, which deals with restricting both the ability to modify data and the ability to *read* data. Since read requests generally do not impact objects/workflows and (in the classical sense) cannot violate integrity constraints, we do not address this topic.

However, we note that if a database system is “logging access requests,” i.e. a database is maintaining a history of outcomes to read/write permissions requests handled by the underlying access control engine, then implementing temporal restrictions based on a sequence of reads/writes is plausible. A business scenario that could justify such a feature is one that requires separation of duties: “if a user has accessed certain parts of the database or made certain modifications in the past, deny the current access request.”

6.2.3 Metric Temporal Logics

Metric Temporal Logics (MTL) or real-time logics are natural extensions of temporal logic where operators are enhanced to include timed restrictions (Figure 6.3). Whereas LTL referred to time implicitly (previously, next, etc.), MTL allows for precisely timed conditions (within some t time periods). A business specification requiring that “a payment must be made within 7 days of approval” is one that lends itself to be modeled in metric temporal logic. Extended operators in such logics operate on a well-defined notion of the domain

of time. MTL operators also look similar to LTL operators, for example \blacklozenge_x and \bullet_x are typically interpreted as sometime in the past x time units and exactly x time units ago, respectively.

However unlike propositional temporal logic and CLTL, there are many different metric temporal logics that have been proposed to cater for very specific situations. For example, in metric interval temporal logic (MITL) the operator $\blacklozenge_{[x,y]}$ can be used to specify an interval of time in the past (between x and y time units in the past) in which a condition must hold true. Similarly MITL can be further extended to include windowed periodicity, and the MPITL operator $\blacklozenge_{[x,y,z]}$ can be used to specify that a condition must hold true for a time period of $y - x$ starting at every z^{th} time period in the past. The business use cases for such complex metric temporal logics are rare. Even though in our search for examples we did not encounter workflows that specifically employed complex metric logics, we can hypothesise that they do manifest themselves in situations where calendar-style time periods need to be identified. For example, a business may want to identify a recurring set of dates in the future to hold meetings, such as the 27th day of every third month unless it falls on a Sunday. Similarly a business may conduct an inventory audit during the second last week of every quarter, and thus specific procedures for introducing new inventory might need to be followed during that time period. Disclosure rules during the “quiet” period before sensitive financial information is released is another example of a recurring business situation that might require different database-level rules to be enforced. One can easily imagine many more examples.

Fortunately, the problem of satisfiability for many fragments of metric temporal logic is no more difficult than the satisfiability of non-metric temporal logic. The key lies in recognizing that a metric operator over k -time periods can be encoded as k -nested applications of non-metric temporal operators. Similarly, if time is treated as a persistent integer t (as is the case in many computer systems), one can encode CLTL formulas that explicitly refer to t to represent MTL formulas. For a comprehensive discussion of these encodings, the reader is directed to the very recently published works of Pradella et al., in which they describe how an existing bounded model checker / SAT solver can be used check the bounded satisfiability of MTL formulas [125, 126]¹. For a recent survey summarizing the

¹Pradella et al. are also the authors of the Zot BMTL solving toolkit [124]

known decidability and satisfiability results for various fragments of MTL, the reader is directed to the work of Ouaknine et al. [118].

Moving into the Future

Although the infrastructure of metric temporal logic fits nicely into the notion of specification and verification of constraints, the restrictions imposed by the original setup still hold. Notably, constraint specification must be restricted to past metric temporal operators only. This is to avoid real-time constraints that are un-enforceable, for example, a situation where “something must happen within x time units into the future” and the database system just sits idle/offline causing a trivial violation through the passage of time. Future metric operators can of course again be used during verification to specify conditions that need to be tested against a system of enforceable constraints specified on the past.

6.2.4 Constraint Systems for Auditing Only

Our final application is auditing in place of constraint enforcement. This is a trivial application having two major benefits. It imposes minimal transactional overhead (i.e., no run-time monitoring costs in addition to keeping an audit log, which most mission critical processes do anyway) and requires extremely little effort to implement by internal/external auditors. DBAs, programmers, and all users/applications can continue to interact with the database under any other constraint regime. However, periodically a copy of a database (that contains a history of all artifacts) is taken and run against various process-centric constraint systems to check for any violations in the past.

In other words, all that is done is to check periodically if constraint violations have already occurred or not. By doing so, violations are not prevented and neither is there a proof that they can never happen, but an organization can, in theory, periodically check its current database for compliance and remedy the situation.

In addition to the above, this process can be made part of a formal audit by an external compliance officer. The application greatly simplifies the actual job of auditing violations.

Instead of having to write complex temporal queries over the database schema to check if specific types of violation occurred, these queries can be built visually using the state-to-state constraint mapping notation that we have presented. Thus, we propose that constraint systems can also be used as a stand-alone query building and database auditing tool.

6.3 Converting Workflows into Constraint Systems

So far, we have asserted the ability of a CLTL based constraint language, such as DCML (or a variant thereof), to capture the control flow semantics of a large class of business processes. In this section we analyze a taxonomy of common control flow constructs found in workflow languages and show how their behaviour can be emulated in CLTL. Thus, we demonstrate that the features present in most modern workflow languages can be easily accommodated in our proposed LTL-based constraint modeling language.

The definitive work on analyzing common “patterns” exhibited in workflows was done by Van Der Aalst et al. [163], who identified 20 patterns pertaining to the flow of control present in many commonly used workflow languages. Since the publication of their work, these patterns have become the gold standard by which workflow languages (visual or lexical) are compared. The reader familiar with LTL and these patterns will be well aware that LTL is a far richer specification language than visual workflows and may find the following reductions trivial. In the subsequent discussion we will refer to these patterns by their names and abbreviations (P1 through P20 [163]).

6.3.1 Basic Control Flow Patterns

Pattern P1 (sequence) requires that a task is completed before another starts and we have already discussed two variants of this pattern in our examples (exit and entry restrictions in Table 3.1 on page 65). Patterns P2 (parallel split), P4 (exclusive choice), P6 (multiple choice) and P16 (deferred choice) are essentially dependencies between 1 state and N

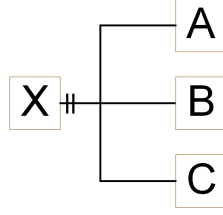


Figure 6.4: A one-to-n pattern specified in DCML where n=3. The constraint can define a dependency between the state X in the previous timestamp and the presence in any/all of A, B, and/or C in the next timestamp.

other states as depicted in Figure 6.4. For parallel split, we can interpret the constraint as the LTL formula as:

$$\bullet X \Rightarrow A \wedge B \wedge C \tag{6.1}$$

Exclusive choice (P4) from X to A, B, or C but not any two of them together can be described in LTL as follows:

$$\bullet X \Rightarrow (A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge C) \tag{6.2}$$

Similarly, multiple choice (P6) is essentially any combination of A, B, and C in the post-condition and equivalent to exit-restrictions in DCML. Deferred choice (P16) is intended to ensure that the process moves on in other respects and the choice between any combination of A, B, and/or C is not made immediately. Since our framework allows an artifact/process to be in multiple states at one time, the object can stay in state X and otherwise progress outside the choice framework of any multi-choice patterns.

Note that some workflow languages can implicitly require that all states/stages be mutually exclusive. Constraints on state transitions in such cases can be accommodated by ensuring that formulas are interpreted slightly differently. The correct LTL equivalent of such workflows languages would be $\bullet precondition \Rightarrow \neg precondition \wedge postcondition$, ensuring that the process moves out of a state when a “choice” at a workflow decision

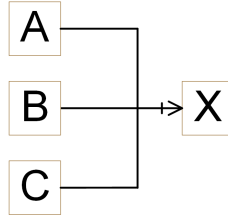


Figure 6.5: A generic n -to-one pattern where $n=3$. The diagram is a mirror of Figure 6.4. The notation can be introduced in DCML to represent constraints of the type specified in Equations 6.1 or 6.2.

point is made. Also note that the examples above (Equations 6.1 and 6.2) only have three “exit states” or choices and that it is trivial to extend them to choices between n -states. Pattern P17 (interleaved parallel routing) is essentially a special case of ordering several tasks, where some can be run in parallel but others need to be completed sequentially. Once again P17 can be accommodated by chaining together several pre-condition implies post-condition style LTL restrictions.

Patterns P3 (synchronization), P5 (simple merge), P7 (synchronization merge), and P8 (multiple merge) are essentially n -to-one constraints in LTL (depicted in Figure 6.5). These can be captured in LTL by simply rearranging the post-conditions and pre-conditions described earlier. For example, a simple merge can be described as:

$$X \Rightarrow \bullet(A \wedge B \wedge C) \tag{6.3}$$

The purpose of different types of synchronization patterns is to accommodate various choice patterns described earlier and to model different wait/proceed requirements. Pattern P9 (discriminator) is closely related to sync/merge patterns, in that it allows workflows to proceed further after any single pre-condition is met. It too is simply modeled by meeting any of the conditions through n OR-clauses.

Pattern P10 (arbitrary cycles) is directly supported because an artifact can cycle back and forth between any two states. In the characterization of Van Der Aalst et al. the issue of bounds in such cycles is not addressed. As mentioned in Section 6.1.3, our modelling

language can be extended to support bounded loops and counters via the use of temporal operators in state definitions. However, the standard DCML presented in Chapter 3 only allows for cycles between states without bounding the number of such cycles.

Pattern P11 (explicit/implicit termination) is supported as well. Either an artifact can transition to ϕ explicitly (from potentially any other state) or simply forever remain in a state that the policy designers can designate as a terminating/archival state. The support for patterns P19 (cancel task) and P20 (cancel case) can also be described in a similar fashion, because any particular branch in the constraint system taken by an object can be abandoned (i.e., does not proceed further) or cancelled (explicitly terminated) at any time. That is, an object can reach a conclusion for every split/choice made or not progress towards any particular path taken, thereby effectively cancelling the subtask in a larger workflow.

Pattern P18 (milestone) is a check condition in workflows that forms a checkpoint for further processing, ensuring that regardless of the prior workflow behaviour, the condition specified in the milestone should be met before proceeding further (see Figure 3.14 on page 76 for an example). The term “milestone” in workflow patterns is used synonymously with a “condition,” and thus any state in a constraint diagram directly represents a milestone.

6.3.2 Multiple Instance (MI) Patterns

In the characterization of workflows proposed by Van Der Aalst et al., there are four patterns that pertain to multiple instances: P12 (without synchronization), P13 (with a priori design-time knowledge), P14 (with a priori run-time knowledge), P15 (without a priori run-time knowledge). Since the intent of these patterns was to capture semantics where the independent processes interact (can be spawned and joined) within a single workflow, it is difficult to associate them directly with constructs in our language, which is aimed directly at restricting the lifecycle of a single artifact type. We do, however, argue that when these patterns are interpreted in a broader sense (as opposed to just threads of execution), many of these patterns are applicable in constraint systems.

Foremost, we note that even though a constraint system describes lifecycles of artifacts, many different artifacts can concurrently trace different parts of this lifecycle. So if

Pattern	Name	Supported
P1	Sequence	Yes
P2	Parallel Split	Yes
P3	Synchronization	Yes
P4	Exclusive Choice	Yes
P5	Simple Merge	Yes
P6	Multi-Choice	Yes
P7	Structured Synchronizing Merge	Yes
P8	Multi-Merge	Yes
P9	Structured Discriminator	Yes
P10	Arbitrary Cycles	Yes
P11	Implicit Termination	Yes
P12	Multiple Instances without Synchronization	Yes*
P13	Multiple Instances with a Priori Design-Time Knowledge	Yes*
P14	Multiple Instances with a Priori Run-Time Knowledge	No*
P15	Multiple Instances without a Priori Run-Time Knowledge	No*
P16	Deferred Choice	Yes
P17	Interleaved Parallel Routing	Yes
P18	Milestone	Yes
P19	Cancel Task	Yes**
P20	Cancel Case	Yes

Table 6.1: A summary of the capabilities of our model under the workflow patterns framework as proposed by Van Der Aalst et al. [163]. *P12-P15 have to be interpreted in the non-traditional sense to be applicable. **P19 is interpreted as a task within the same workflow/canvas

an artifact contains within it n sub-artifacts, we could potentially model the lifecycle of individual interactions of these n objects.

In our examples of invoices and sales orders, these lifecycles were completely independent and could not “communicate” with each other (e.g., one invoice could not “share variables” with another, as done by concurrently running threads). Thus, on the surface it may seem as if multiple instance patterns cannot be supported. However, as mentioned above, we can broaden the description of an artifact by including within the definition interactions between a bounded number of similar process-oriented concepts. For example,

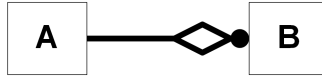


Figure 6.6: Visual notation (lines, boxes, etc.) serve as templates for various LTL based control flow restrictions in a workflow. For example, the above newly introduced arrow could be interpreted as an LTL restriction specifying that “A implies always in the past B” with its logical interpretation as $A \Rightarrow \blacksquare B$. Such constraints cannot be modelled in visual workflow languages used in everyday business situations and thus require more intimate knowledge of LTL on the part of policy makers.

we can model “pairs of invoices” as artifacts (via a self join) and thus argue that with a priori knowledge about the number/type of concurrent sub-processes ($n=2$ when examining all pairs), we can construct complex workflows that model interactions between multiple same-type artifacts.

Building a constraint system for these oddly interacting artifacts is non-trivial, but the end result does somewhat conform to the notion of reasoning with multiple instances with a priori design time knowledge. We still cannot “spawn” tasks within workflows in the traditional sense, since that would require taking actions on the database, and, as discussed earlier, that functionality is beyond the scope of this work. Thus, we argue that in some circumstances we can accomplish a priori reasoning over multiple instances. We also note that ours is not the first instance where these patterns have to be interpreted such that they can be accommodated over a proposed modeling language. Prior work has had to make similar interpretations on what such MI interactions would mean in a scenario that is not directly relatable to traditional pictorial workflows [67].

6.3.3 A Note on Notation

In this chapter, we have discussed several ways in which our originally proposed DCML semantics (Table 3.1 on page 65) can be extended to accommodate the needs of policy makers in a broad range of scenarios. The extensions for which we specifically depicted visual notation include metric temporal constraints (Figure 6.3 on page 143) and temporal access control constraints (Figure 6.2 on page 140).

We wish to point out that the purpose of proposing pictorial representations of such constraints was make a broader point that an LTL/state-oriented constraint system is an extremely flexible tool that can be adopted to suit the needs of individual policy makers in a large number of situations. Since DCML is very closely related to LTL, a wide range of constructs can be introduced into the modeling language to extend DCML. Figure 6.6 above presents one final example of such visual constraints.

In this work we deliberately choose not to present and promote a definitive all encompassing extended DCML/LTL constraint system toolkit. There are two major reasons why we refrain from suggesting the holy grail of LTL constraint system restrictions and only discuss possibilities for extending DCML. Foremost is the fact that visual notation is best left for policy designers (or graphic designers) to create. Choosing the right arrow-heads, type of lines, and colours to use is best left for the users that work in the trenches of workflows and corporate policy management. Consequently, if users of our proposed technique want to re-interpret different arrows for their own needs, they are free to do so. Secondly, it would go against the principles of extensibility, customizability and, separation/distribution of responsibilities to propose a standard one-size-fits-all set of state-oriented restrictions and call it a universal toolkit. Examples such as Figure 6.6 above have repeatedly shown that there is a lot LTL can do which goes beyond traditional workflows, and thus there should be room for database level constraint system designers to extend their toolkits according to the specific situation that they are in and to tailor it for specific audiences. Our main goal in introducing fixed semantics for DCML in this thesis is to demonstrate the overall usability of an LTL based visual constraint modeling language.

6.4 Summary and Conclusions

In this chapter we examined several issues that lie at the intersection of usability and limitations of our proposed constraint modeling framework. We observed that there are three key factors that need to be considered when attempting to extend the proposed infrastructure: artifact type complexity (view complexity), constraint system complexity, and the complexity of the property that warrants verification. We discussed three specific

extensions and examined how each of these factors played a role in enforcement and verification. Metric temporal logic [8] was an obvious first choice to examine since it is directly supported by modern LTL model checkers / SAT solvers and very intuitively provides additional expressiveness in temporal logic. Next we examined the problem of specifying process-oriented temporal access control restrictions [111, 129]. We showed that transaction related meta-data can be made an integral part of the artifact and thus can be used to rephrase access control restrictions as restrictions on the history/evolution of an artifact. We pointed out limitations that must be adhered to and caveats for DBAs verifying properties on and across these enhanced artifact constraint systems. In our final example of usability, we argued that the visual depiction of constraint systems can be used for auditing (i.e., building auditing queries) requiring very little buy-in from system administrators.

Along the way we hope to have made clear several important points about usability and extensibility. Our framework can easily accommodate the majority of workflow patterns used in modern workflows and can go well beyond simple flow-oriented restrictions, especially in situations where various extended fragments of LTL are employed together. Even within one fragment, a wide array of restrictions that do not correspond to traditional workflows can be modelled. We pointed out that with such flexibility, it is unlikely that a single set of restrictions can accommodate the needs of all policy makers. Thus, the task of proposing a definitive set of visual restrictions on a drawing canvas is best left to DBAs and corporate policy makers for their specific situations. We believe that there are parallels between our proposed modelling language and meta-modelling languages presenting an approach that can be tailored to specific applications. We believe that like-minded user groups and policy makers should thus be allowed to create and share various customized toolkits along with interpretations of constraints for their specific needs.

Chapter 7

Case Study: Form GC179

In this chapter we present a real-world business process and examine in detail how complex operational constraints can be translated from their visual and textual descriptions into DCML models. Section 7.1 provides the necessary background and identifies the source documents from which policy specifications are taken for this case study. It also highlights our aims, objectives, and assumptions. In Section 7.2 we present in detail the business process being examined. Section 7.3 then presents our interpretation of the business process in DCML. Section 7.4 generalizes the specific case study presented in this chapter and highlights several variables that were either not fully captured by the example presented or cannot be captured in one case study alone. It presents a comprehensive cost-benefit analysis of our proposed constraint modeling, enforcement, and verification infrastructure which can be used to measure its true usefulness. Finally, Section 7.5 concludes this chapter by providing a brief summary of this chapter.

7.1 Background

The payroll responsibilities for the Government of Canada are handled by the department of Public Works and Government Services (PWGS). The Regional Pay System (RPS) operated by PWGS administers the payroll for more than 300,000 employees of the federal

government within 110 departments and agencies. The system operated by PWGS is the largest payroll deployment in Canada, and it performs approximately 8.9 million financial transactions annually valued at roughly \$17 billion Canadian Dollars [127].

One specific function of this payroll system is to interact with various departments and handle requests for extra pay made by the employees of the government. Form *GC179* (also known as the *The Extra Duty Pay/Shiftwork Report and Authorization Form*) is the designation given to a formal request for additional salary (typically in lieu of overtime services rendered) made by an employee of the Government of Canada. For various reasons, including security, privacy, and accessibility, each department of the government can design, maintain, and manage its own GC179 form/template. The only major requirement in handling such forms is that the final request for cash payments must be exported into the RPS operated by PWGS Canada. We speculate that for most departments of the federal government, GC179 requests are completed and processed electronically. However, examples of conditions under which flexibility in handling GC179 requests is required include situations where employees are unable to access a paper form or a computer (such as in remote field operations). Some departments of the government offer alternative means of handling GC179 requests for their employees. For example, the Department of Finance allows its employees to accrue vacation time instead of receiving taxable cash salary for the overtime service rendered [55]. In short, there is no standardized GC179 form or associated process across the federal government.

Consequently, each department of the Government of Canada must design its own set of policies, rules, and processes for handling GC179 requests. It stands to reason that the various HR related databases operated by the departments of the government must be subject to certain rules developed internally. Most importantly, each department is subject to audits conducted by other branches of the government, such as the Treasury Board of Canada (TBC) and the Office of the Auditor General. Naturally, such audits scrutinize the published practices of every department and verify that the data regarding pay administration complies with the published practices. In addition, the branches of Government that oversee and audit the operations of others' also publish their own interpretations of legislation in the form of policy directives and guidelines on how to handle various processes. For example, the policy suite published by the TBC [155] contains a host of policies that each

department of the government should consider before developing its own internal rules. The suite includes recommendations and guidelines on many employee related issues, such as fairness in pay and working conditions, as well as handling of financial matters when it comes to employees. Although departments can use these policy suites as guidelines, they are not specification documents delineating specific business processes, and therefore there is no single policy document that specifies how a GC179 request should be handled.

The challenge of compliance with overtime-pay related rules for a department of government is three-fold: (1) design a coherent and workable set of rules to handle GC179 requests, (2) ensure that the rules do not conflict with the overall policy directives of the Government of Canada, and (3) ensure that the record keeping aspect of the department (database) is compliant with the designed rules. The case study presented herein examines the above three aspects of compliance in handling GC179 requests within one specific department of the Government of Canada.

7.1.1 Purpose of the Study

Our goal in this chapter of the thesis is to highlight the usability and usefulness of our proposed constraint modeling language (DCML) and the associated verification infrastructure. We do so by accomplishing two specific objectives: (i) demonstrating that a typical business process (in our case represented by the handling of GC179 requests), when codified by business users in terms of process models, flowcharts, and associated textual descriptions of the process, can be easily mapped onto a DCML diagram; and (ii) demonstrating that the implementation and verification of DCML constraint models that represent real world situations can be accomplished by users with minimal technical understanding of CLTL.

Thus, we hope to show that the framework presented in this thesis is practical, efficient, and easy to use for database systems that store information about real business processes. We believe that the chosen example is representative of a large class of sequential business processes. To further support our claims, the workflow and associated constraints presented are taken directly from publicly available sources without significant alteration.

7.1.2 Information Sources and Assumptions

Many departments of the government regularly publish the findings of internal and external audits. For the specific case of handling GC179 requests, several departments, including Veterans Affairs Canada, Agriculture and Agri-Food Canada, and the Department of Finance Canada have made their GC179 audit reports and findings publicly available [6, 55, 168]. Typically, these reports serve the purpose of increasing transparency. However, the information contained in these audits about the processes and specific safeguards that should (and are) in place represents an ideal testing ground of converting process-centric business rules onto database level constraints.

In this thesis we focus on the specific case of the Department of Finance, which, among many of its duties, is responsible for preparing the budget of Canada and handling the financing/borrowing operations of the Government of Canada in financial markets. Our primary reason for choosing the Department of Finance's audit report for handling overtime pay requests [55] is that it is the most comprehensive of its kind. More specifically, it (a) explicitly lists and details the various checks and constraints on the processing of GC179 requests, (b) provides rationale in developing constraints by linking them to high level policy objectives as published by the Treasury Board of Canada, and (c) contains visual/graphical depictions of the business processes associated with GC179 requests.

In short, the report comes very close to being a specifications document created by the department, by virtue of it containing most of the information required to convert business processes into database level constraints. The only significant aspect of information missing from the report is the description of a database schema (table structure and data fields) of the HR database of the Department of Finance. Therefore, the only major assumption on our part is that of extracting a list of data fields and attributes (more specifically, an artifact type definition) relevant to the processing of GC179 requests based on the details presented in the report.

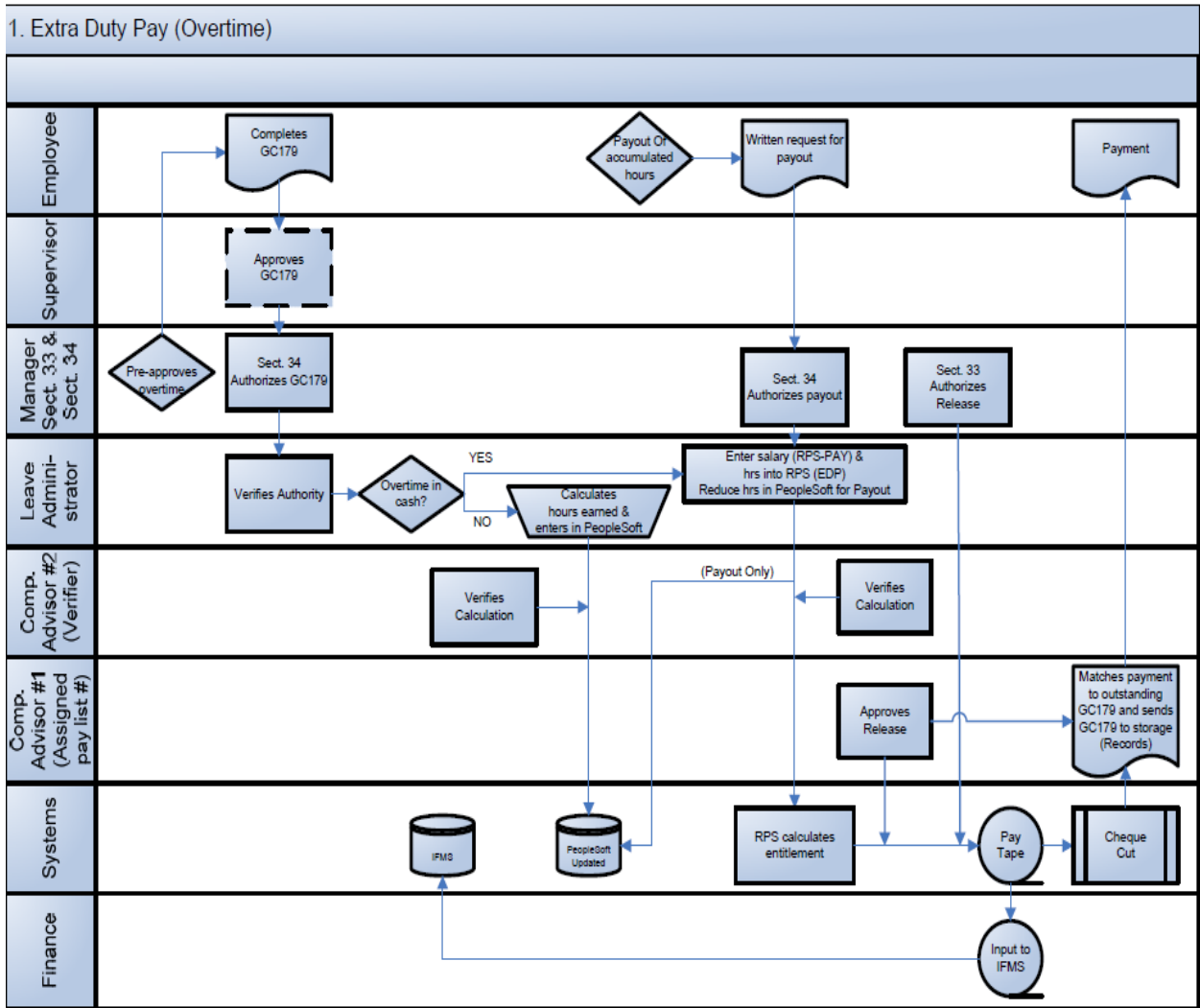


Figure 7.1: GC179 overtime request handling process model taken from Appendix 2 in Department of Finance Audit of Leave and Overtime Report [55].

7.2 Process Model and Flowchart

The process model for handling GC179 requests, taken directly from Department of Finance Audit of Leave and Overtime Report [55], is presented in Figure 7.1. The model can be traced from top-left, where an employee completes a GC179 form, to the bottom-right, where data is exported into the RPS and a cheque is cut for the employee. The description of this process and specific constraints are summarized below.

1. All GC179 requests are handled electronically and can be created in the system in two ways. After completing the overtime work, an employee can either complete a GC179 request or the same can be completed by a designate of the employee. In case a designated employee files the GC179 request, it must subsequently be reviewed and cross-checked by the employee named on the GC179. This additional step must be done electronically as a discrete and separate event. However, there is no explicit requirement that the actual transaction that marks a GC179 request as cross-checked is performed by the employee named on the request. GC179 requests can be marked as cross-checked by the designate, as long as this is done in a separate transaction. The purpose of the additional step is to ensure (at least in theory) that the designate actually confirms the details with the employee *after* creating a GC179 request.
2. All GC179 requests require supervisor approval before they can be processed. Even if the GC179 request was pre-approved, it must be reviewed to ensure that the scope of the request matches that of the pre-approval. In addition, an employee can not be his or her own supervisor, and once a supervisor approval is given, the details of the request can no longer be altered by the employee unless the request is explicitly “sent back” to the employee for amendments. Finally, a supervisor can only approve, permanently reject, or send back the request and can not alter it in any other way. If amendments to the request are required they must be made by the named employee or his/her designate in a manner described in point (1) above.
3. GC179 requests that have received supervisor approval must then be authorized by an employee that is the Section 34 Authority in the department, as designated in the Financial Administration Act of Canada (1985). The designated Section 34 Authority

for a department are one or more employees that are responsible for the overall financial oversight. The purpose of this approval is to ensure that the charge/claim is properly documented, reasonable, and necessary. The employee acting as the Section 34 Authority must be different from both the employee named in the GC179 request and the supervisor of that employee. Once again, the Section 34 Authority can only approve, permanently reject, or send back the request and can not alter it in any other way. If amendments to the request are required, they must be made by the named employee or his/her designate in a manner described in point (1) above.

4. GC179 requests that receive Section 34 approval are then processed by an HR administrator of the department. Employees can be compensated for overtime in two ways: cash or vacation hours (eight vacation hours are treated as an extra day of vacation). Employees must choose at the time of completing a GC179 request how they wish to be compensated.
 - (a) If an employee requests vacation hours, a calculation is made by the HR staff to determine how many vacation hours should be accrued. This is done on a case-by-case basis because the multiple applied to overtime hours to determine vacation hours can change depending on circumstances (e.g., overtime pay rules may be different for regular weekends and statutory holidays). The final number of accrued hours calculated is verified by a comptroller of records and then added to the total number of available vacation hours on record for the employee.
 - (b) If cash payment for overtime is requested, then the HR staff calculates (based on the employee's pay rate or yearly salary) the appropriate amount to be paid for this GC179 request. This calculation is verified by a comptroller of financial records and then made ready for export into RPS. Once RPS has processed the payment, it returns a payment identifier, which is then associated with this GC179 request, and no subsequent changes are allowed to the request.
5. GC179 requests that have not yet been processed by HR can be deferred back to the employee or designate for changes. This is typically done when any of the supervisor, Section 34 Authority, or the HR staff is not satisfied with the request and its associated details. When a request is deferred back to the employee it loses its supervisor

and Section 34 approvals and must go through the process (including cross checking if necessary) once again.

6. GC179 requests that have not yet been processed by HR can also be permanently rejected. Once a GC179 request is rejected, it can not be modified in any way. Therefore, all processing on such requests is terminated and they are forever archived.
7. There is also an alternate process for employees who have accumulated a large amount of vacation hours to “cash them out.” A written request must be received from the employee, which must then be scrutinized by the Section 34 Authority (for availability of funds and reasons for the request), and then subsequently forwarded to HR. An employee in HR will re-examine one or more prior GC179 requests for which vacation hours were awarded, subtract the vacation hours from the employee’s remaining vacation hours, perform the appropriate salary calculations, and make the salary data ready for export into RPS. Before the data is exported into RPS and a cheque is cut, it must be verified by a comptroller.

7.3 Implementation

7.3.1 Artifact and State Definitions

Since the audit report does not specifically mention the internals of the database operated by the Department of Finance, we assume that the attributes listed in Table 7.3.1 define the GC179 artifact type completely. We believe that most database designers will treat a GC179 request as a strong entity, and thus the artifact definition will only consist of a selection on a single table. Constraints (DCML diagrams) will therefore be applicable on a single sequence of assignments to these attributes.

From the description of the process given in Section 7.2, we extrapolate that, at a high level, there are 3 different phases in the lifecycle of a typical GC179 request: (i) data entry, (ii) managerial approvals and (iii) HR/back-office processing. Based on our reading of the process, we have identified 13 distinct states involved in the processing of GC179 artifacts

Attribute Name	Type	Details
GC179_ID	int	Unique identifier of the request
EMP_ID	int	Employee ID of the beneficiary
FILLED_BY_EMP_ID	int	Employee ID of the designate (if any)
CROSS_CHECKED	bit	Request cross-checked by the beneficiary?
DETAILS	varchar	Details associated with the request
CHANGES_REQUIRED	bit	Does request/form data need editing?
APPROVAL_SUPERVISOR_EMP_ID	int	Employee ID of the supervisor
SEC34_AUTHORITY_EMP_ID	int	Employee ID of Sect. 34 Authority
HR_EMP_ID	int	Employee ID of the person who performed the vacation/salary calculation
HOURS_ACCRUED	float	Vacation hours credited in lieu of overtime
CASH_PAYMENT_AMOUNT	float	Cash paid in lieu of overtime
HR-PAYROLL_TXN_ID	int	Transaction identifier for the payroll / HR calculation record
EXPORTED_TO_RPS	bit	Request exported into RPS?
PAYOUT_REQUESTED	bit	Was a cash payment requested?
COMPTROLLER_EMP_ID	int	Employee ID of Sect. 33 Authority / Comptroller
PMT_ID	int	Transaction ID of the payment made to the beneficiary by the RPS
LASTUPDATE_EMP_ID	int	Employee ID of the person who last modified the request

Table 7.1: Attributes of the GC179 artifact

DCML State	Explanation and Purpose
ϕ_S	Uninitialized State
FILLED BY DESIGNATE	Request completed by a designated employee and awaiting verification.
SELF FILLED	Request completed by employee (no cross checking required). Awaiting supervisor approval.
CROSS CHECKED	Request cross checked and awaiting supervisor approval.
CHANGES REQUIRED	Request deferred back to employee for amendments.
REJECTED (ϕ_E)	Request permanently rejected (artifact life-cycle terminated).
APPROVED BY SUPERVISOR	Request approved by supervisor and awaiting Section 34 approval.
APPROVED BY SECT34 AUTHORITY	Approved by Section 34 Authority and awaiting HR processing
VACATION HOURS CALCULATED	Vacation hours calculated and request is awaiting comptroller review.
CASH PAY CALCULATED	Cash pay calculated and request is awaiting comptroller review.
DATA EXPORTED TO RPS	Comptroller review completed and data exported to payroll system.
HOURS ACCRUED	Comptroller review completed and hours accrued to employee vacation record.
PAYMENT MADE	Payment confirmation received by RPS.

Table 7.2: States of GC179 artifact

in the above three phases. They are listed in Table 7.3.1. Our interpretation given to the states is very intuitive and augments the visual process model presented in Figure 7.1. In the next section we provide our DCML diagrams for each of these phases.

7.3.2 DCML Diagrams

Phase 1: Data Entry

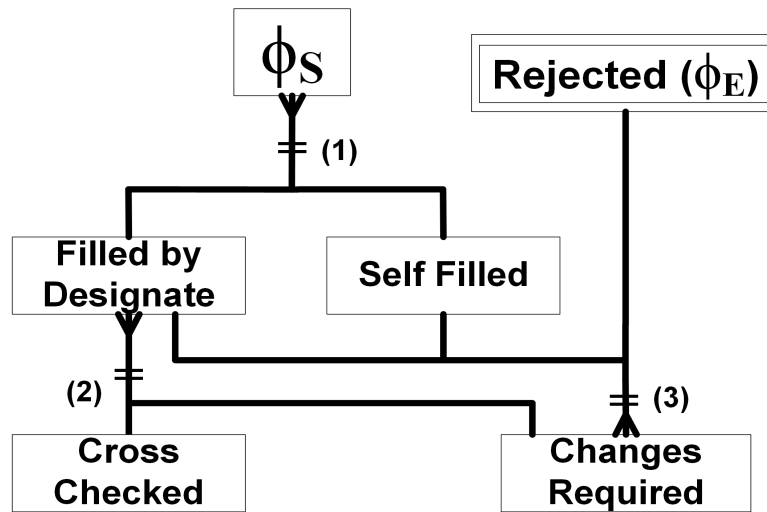


Figure 7.2: Data Entry Phase of the GC179 lifecycle

Figure 7.2 summarizes the data-entry phase of handling GC179 requests in DCML and Figure 7.3 presents it in full with all the states fully defined. There are only three constraints during this phase and they are as follows:

1. Initialization: A GC179 artifact can be instantiated in either the SELF FILLED or FILLED BY DESIGNATE states.
2. Cross Checking: When leaving the FILLED BY DESIGNATE state, an artifact must either enter the CROSS CHECKED state or the CHANGES REQUIRED state.

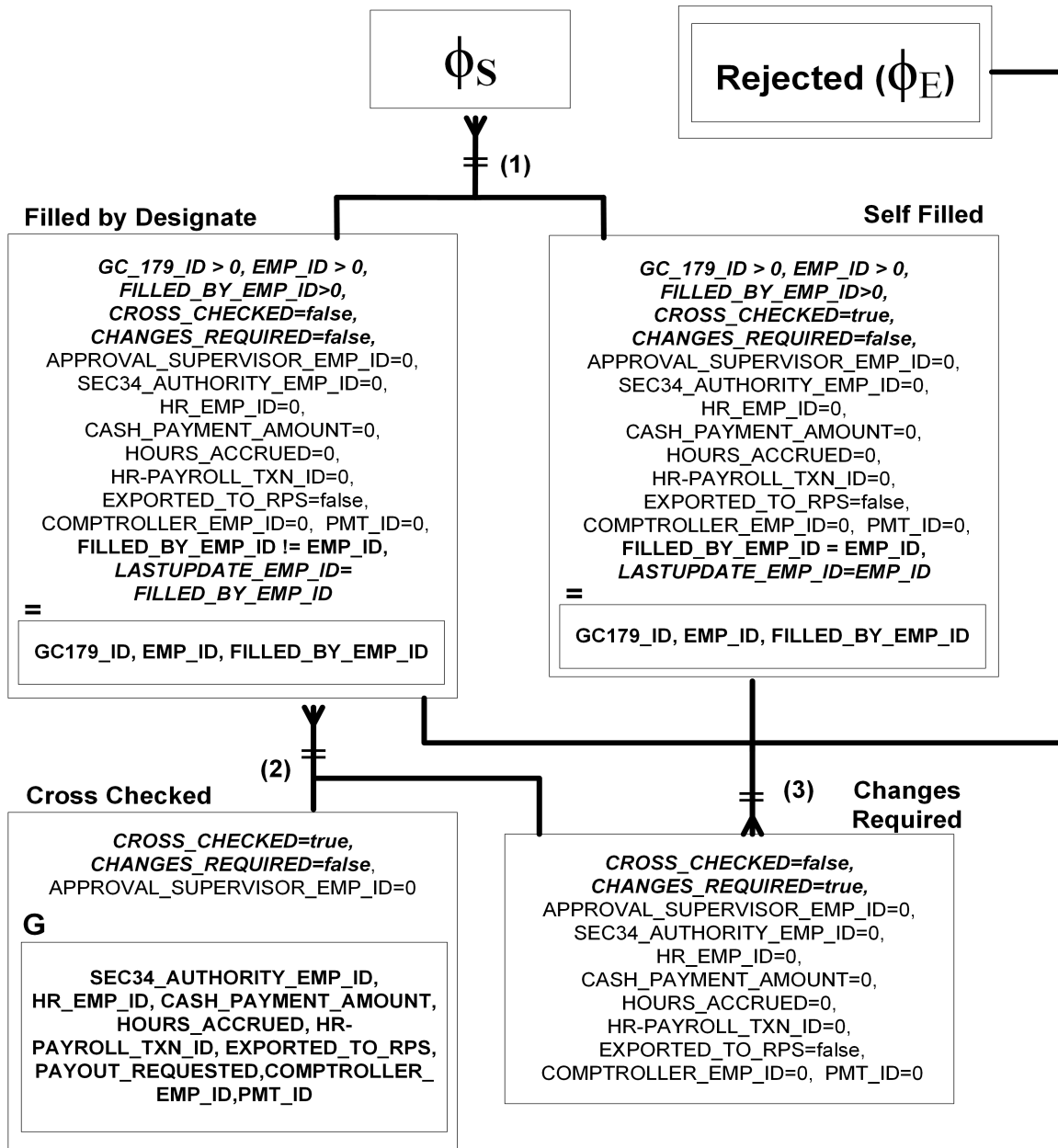


Figure 7.3: Data Entry Phase of the GC179 lifecycle - Complete DCML Canvas. The attributes and their required values that differentiate one state from the others are highlighted in bold font

3. Amending GC179 requests: A request in the CHANGES REQUIRED state, when leaving this state, can only go to the SELF FILLED, FILLED BY DESIGNATE or the REJECTED states.

There are some important observations that need to be pointed out. Foremost, under the designed constraint system a value of *zero* is assumed to be unassigned. For example, when the artifact is instantiated (either in the SELF FILLED or FILLED BY DESIGNATE states), the Section 34 Authority for it is required to be zero. The value of zero is used to imply that the Section 34 approval has not yet been given. When this value becomes positive in the lifecycle of the artifact, then the assigned value will be interpreted as the employee identifier of the Section 34 Authority that approved the request. Note that a similar interpretation is given to the EMP_ID attribute (representing the employee identifier of the person named on the request). In either of the instantiation states, this attribute must be positive, which is then interpreted to imply that an employee ID has been assigned or must be associated with every GC179 request at instantiation time.

Both the instantiation states (FILLED BY DESIGNATE and SELF FILLED) require that upon reaching either of them the attributes GC179_ID, EMP_ID and FILLED_BY_EMP_ID become invariants. Therefore, during the lifecycle of an individual GC179 artifact, these variables will retain the values they are assigned during instantiation.

Also note that the state conditions are defined to ensure that an artifact that is in the SELF FILLED state is also in the CROSS CHECKED state. This is because self-filled GC179 requests do not require an explicit cross checking and thus are ready to move on to the next phase of processing. However, an artifact that is in the FILLED BY DESIGNATE state must explicitly transition to the CROSS CHECKED (via a change in the CROSS_CHECKED attribute). Similarly, the state CHANGES REQUIRED is distinguished by the boolean attribute CHANGES_REQUIRED, and all artifacts exiting this state must have this flag unset before progressing further. Constraint 3 ensures this and accomplishes the business objectives requiring that a GC179 request that has been “sent back” to the employee for amendments must be examined by the employee or designate (to be corrected in some way) before being put back in the processing pipeline.

Phase 2: Managerial Approvals

This phase of the process involves managerial review of the GC179 request, and the only two relevant states in this phase are APPROVED BY SUPERVISOR and APPROVED BY SECT34 AUTHORITY. As before, Figure 7.4 presents a DCML canvas that summarizes the constraints in this phase, and Figure 7.5 fully defines the states.

Note that during this Phase of processing, there are several ways in which the artifact can be “pushed back” in its lifecycle to Phase 1. For example, the request may be deemed to be incomplete or inadequate by the supervisor, by the Section 34 Authority, or by the HR coordinator. In each of these cases, the artifact is moved to the CHANGES REQUIRED state which, the way we have segregated states, is not explicitly part of Phase 2. Consequently, we include some states from Phase 1 to depict fully constraints that take the artifact to the prior phase of processing. A brief summary of the constraints presented in Figures 7.4 and 7.5 is as follows:

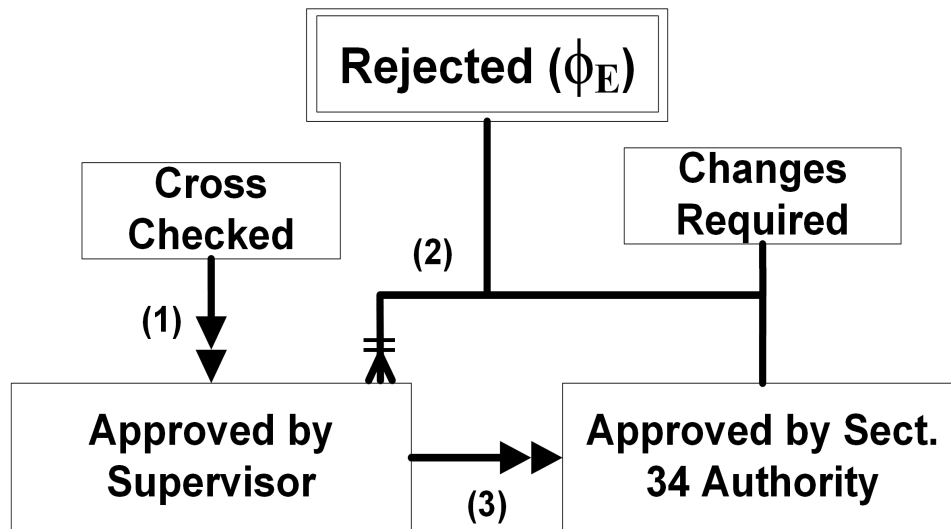


Figure 7.4: Managerial Approvals Phase of the GC179 lifecycle. Two states that were not previously fully defined are introduced and their interaction with prior states is summarized

1. Supervisor Approval: A GC179 request can only arrive in the APPROVED BY SUPERVISOR state if it was previously in the CROSS CHECKED state. Ob-

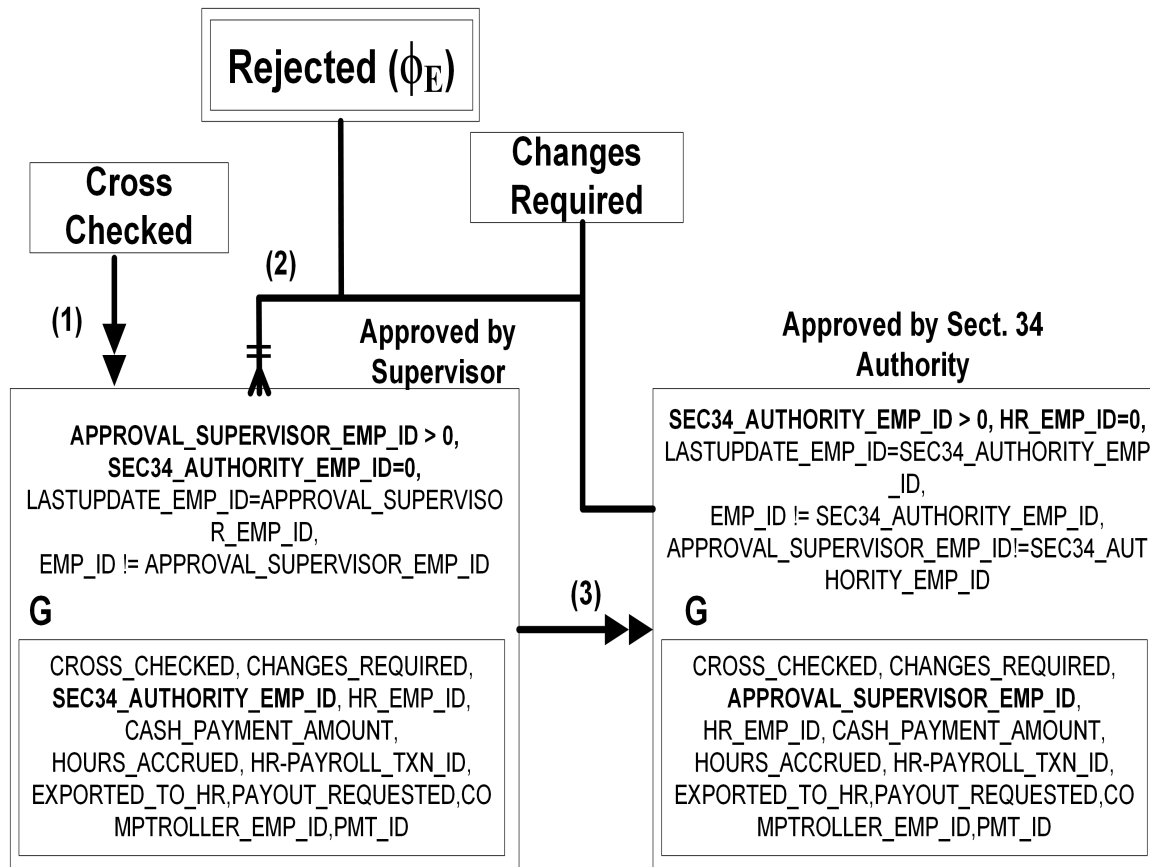


Figure 7.5: Managerial Approvals Phase of the GC179 lifecycle - Complete DCML Canvas

serve from the function defining the APPROVED BY SUPERVISOR state that APPROVAL_SUPERVISOR_EMP_ID must be set to a positive value. In addition, there are constraints on who can modify the artifact (i.e., who can provide the supervisor approval) that ensure that the supervisor is different from the employee named on the request. Finally, to ensure that a supervisor only provides approval and does not materially modify the request, all other variables are guarded (sub-container labelled G) in the APPROVED BY SUPERVISOR state.

2. Leaving the APPROVED BY SUPERVISOR state: Constraint (2) in Figures 7.4 and 7.5 ensures that an artifact leaving the APPROVED BY SUPERVISOR state

must only be able to go to one or more of REJECTED, CHANGES REQUIRED or APPROVED BY SEC34 AUTHORITY states. But since each of these states is non-intersecting, i.e., the functions associated with any two of these states cannot be satisfied at the same time, an artifact can only be in one of them when it leaves the APPROVED BY SUPERVISOR state. Thus, combined with the guard conditions in constraint (1), this further restricts which attributes modifications must take place in order for an artifact to exit towards a particular state.

3. Entering the APPROVED BY SEC34 AUTHORITY state: Constraint (3) is an entry restriction similar to constraint (1). Note that while constraint (2) implied that when leaving the APPROVED BY SUPERVISOR state an artifact *may* go to the APPROVED BY SEC34 AUTHORITY state, constraint (3) requires that if an artifact enters the APPROVED BY SEC34 AUTHORITY state it must have been previously been in the APPROVED BY SUPERVISOR state. The net result of constraints (2) and (3) is somewhat similar to the micro-example presented in Section 3.5 on page 65. Once again the key differentiator of the exit state (APPROVED BY SEC34 AUTHORITY) is that the SEC34_AUTHORITY_EMP_ID must be positive (i.e., assigned) implying approval by the employee responsible for financial oversight in the department. To ensure that the Section 34 Authority only provides the specific approval and does not materially modify the request, all other variables are guarded in the APPROVED BY SEC34 AUTHORITY state.

Phase 3: HR and Back-office Processing

The final phase of handling GC179 requests and the associated constraints are depicted in Figures 7.6 and 7.7. Recall from Section 7.2 that GC179 requests that have received Section 34 approval must go through one final review and approval by an HR coordinator responsible for handling the payroll and vacation records for employees. The calculation determining overtime salary or the equivalent in vacation hours is performed as part of this review. Once completed the request can no longer be rejected and must proceed to payment or accrual of vacation hours.

Constraint (1) in Figures 7.6 and 7.7 restricts what can happen to artifacts that have

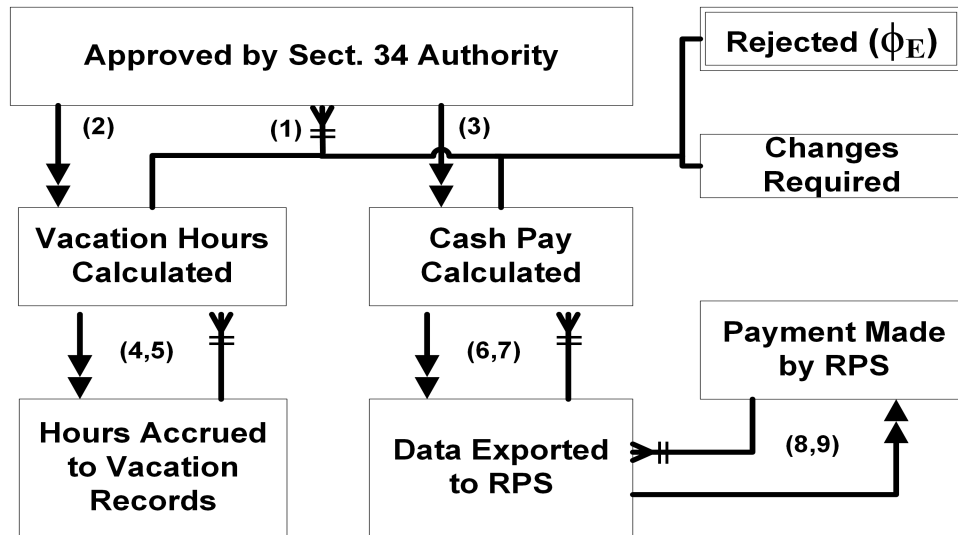


Figure 7.6: HR and back-office processing phase of the GC179 lifecycle.

received Section 34 approval. More specifically, such artifacts can either be permanently rejected, sent back to the employee for amendments, or move forward in processing towards payment or vacation hours being accrued for the employee. Constraints (2) and (3) ensure that if a calculation for vacation hours or cash payment respectively has been completed, then the artifact must have previously been in the APPROVED BY SEC34 AUTHORITY state. Since the states VACATION HOURS CALCULATED and CASH PAY CALCULATED are by definition mutually exclusive an artifact can only progress in one of the directions based on the value of the PAYOUT_REQUESTED flag.

Constraints (4-5) together have the net effect of ensuring that an artifact waiting for comptroller review in the VACATION HOURS CALCULATED state can only move to the HOURS ACCRUED state and an artifact moving into the HOURS ACCRUED state must have previously been in the VACATION HOURS CALCULATED state. This does not preclude the calculation of hours accrued to be corrected or repeated, since the artifact will remain in the VACATION HOURS CALCULATED states if this happens. For example, if the comptroller discovers an error, the HR coordinator can still correct the relevant field of the artifact (HOURS_ACCRUED) to a different non-negative value while ensuring that the artifact remains in the VACATION HOURS CALCULATED. However, both

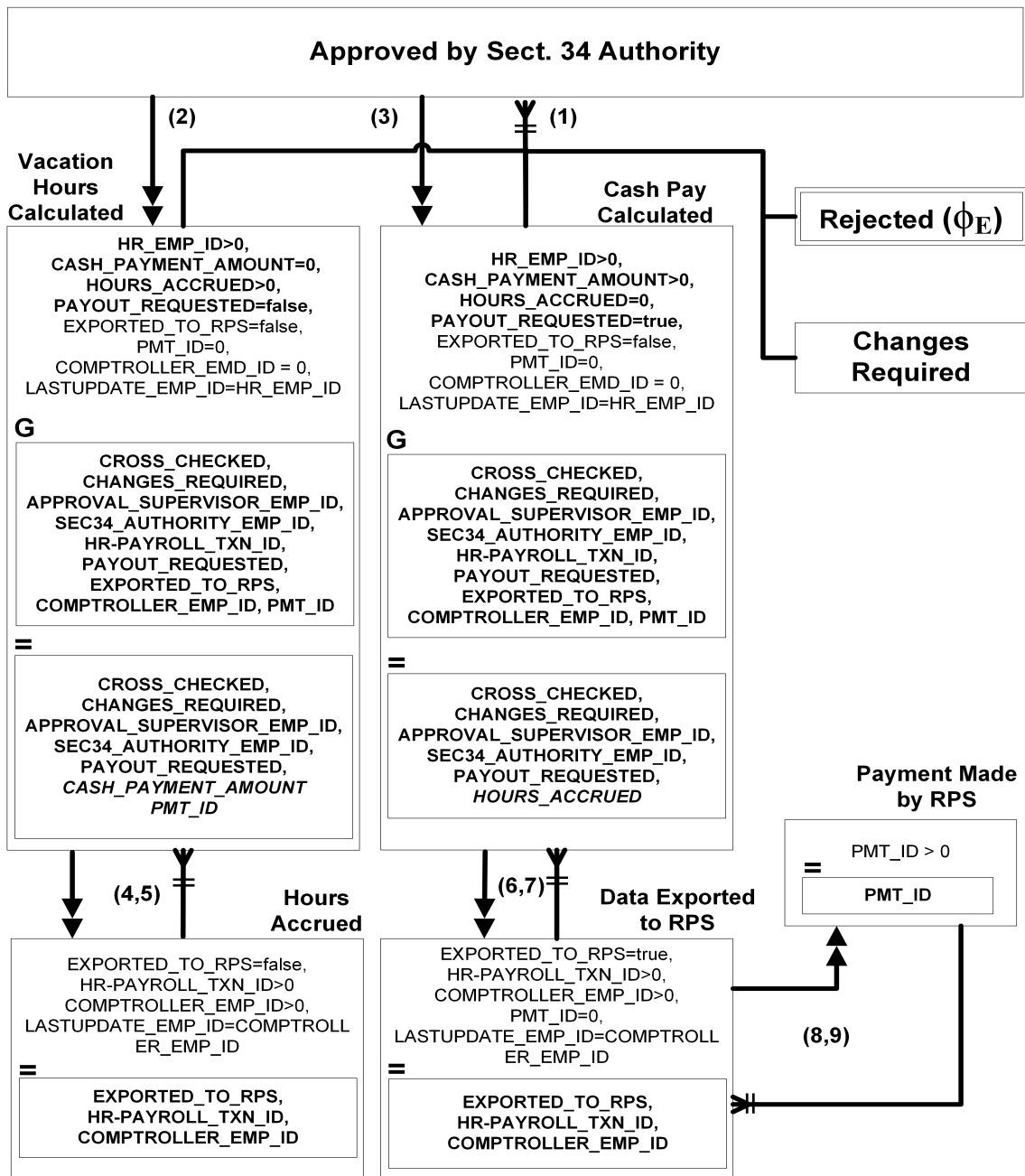


Figure 7.7: HR and back-office processing phase of the GC179 lifecycle - Complete DCML Canvas.

these constraints are necessary to ensure that the path between these two states is strictly followed and the object does not “escape” by going off diagram. Similarly, constraints (6-7) and (8-9) accomplish the same objective of ensuring that an object only transitions between the two states linked by these constraints.

Observe that there is extensive use of guards and invariant conditions in the four major states of interest in this phase. The purpose of guarding a significant number of variables is to ensure that the HR coordinator does not modify critical information that s/he is not authorized to change. Because requests that have passed the last stage of review can not be sent back for amendments, a significant number of attributes are made invariants as soon as the calculation for pay or vacation hours is completed. The key variables that are made immutable pertain to the earlier phases of processing and include those used to determine whether the request needs changes (`CROSS_CHECKED` and `CHANGES_REQUIRED`), the approval supervisors identifiers (`APPROVAL_SUPERVISOR_EMP_ID` and `SEC34_AUTHORITY_EMP_ID`), and the method of remuneration requested by employee. Similarly, once the comptroller review is completed (in the states `HOURS_ACCRUED` and `DATA_EXPORTED_TO_RPS`), additional variables pertaining to processing are made invariants. Finally, in the case a cash payment is requested, the last variable to be modified is `PMT_ID`, which is returned by the payroll system external to the department.

7.3.3 The GC179 Constraint System

The single unified DCML constraint diagram that brings together the three phases of processing GC179 requests is presented in Figure 7.8. The Zot code generated by translating each visual constraint into CLTL is provided in Appendix A. The translation of DCML to Zot was done according to the semantics of DCML presented in Table 3.1 on page 65.

To illustrate how this translation can be accomplished (and automated in the future using a visual modeling tool), we now examine a two-state fragment of the complete GC179 artifact constraint system in further detail. The canvas we consider consists of the `DATA_EXPORTED_INTO_RPS` and `PAYMENT_MADE_BY_RPS` states (bottom right of Figure 7.8) and is presented in Figure 7.9. The definition of the `DATA_EXPORTED_INTO_RPS`

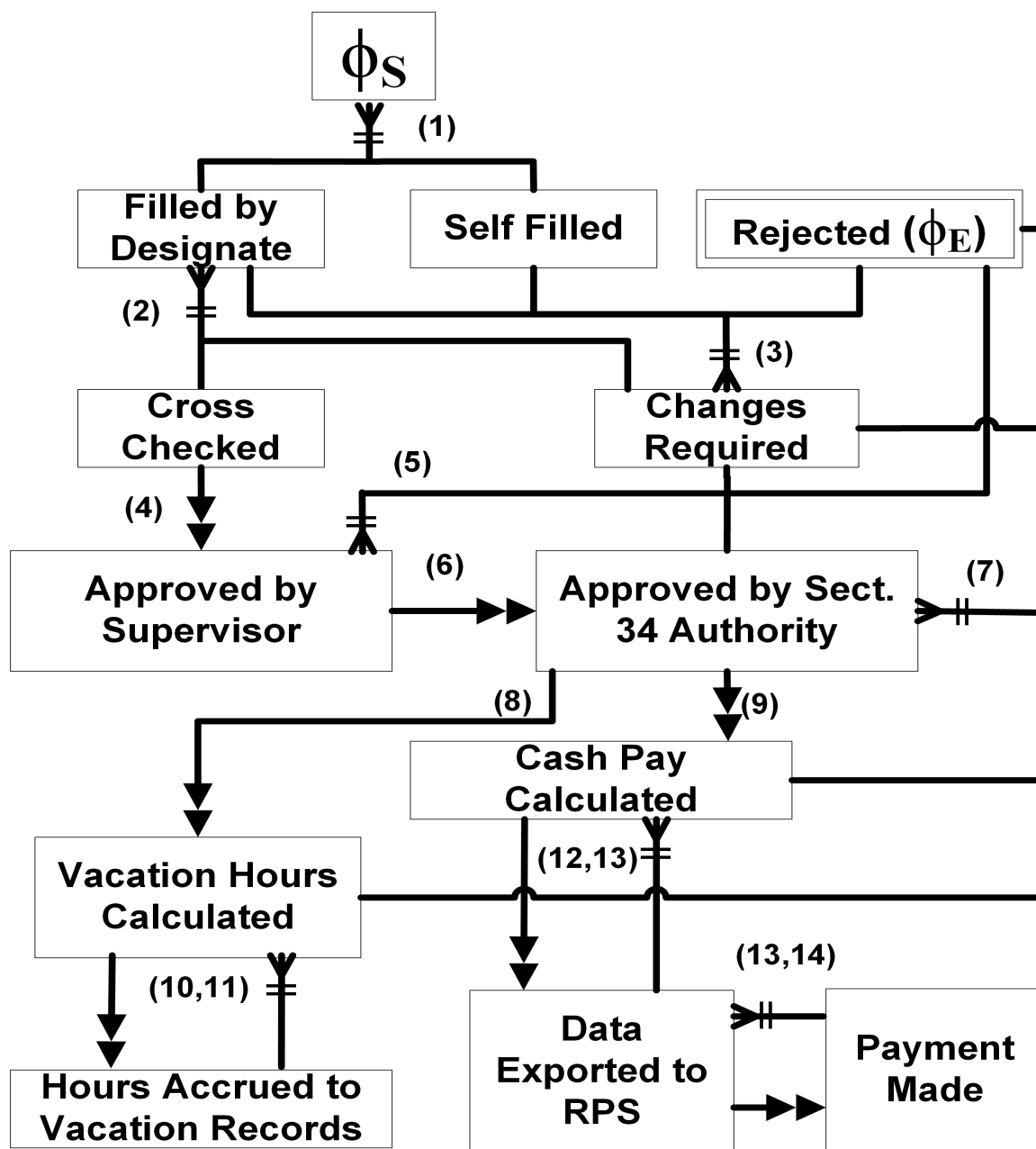


Figure 7.8: Unified DCML Diagram for GC179 requests. See Appendix A for the same model translated into Zot code for verification.

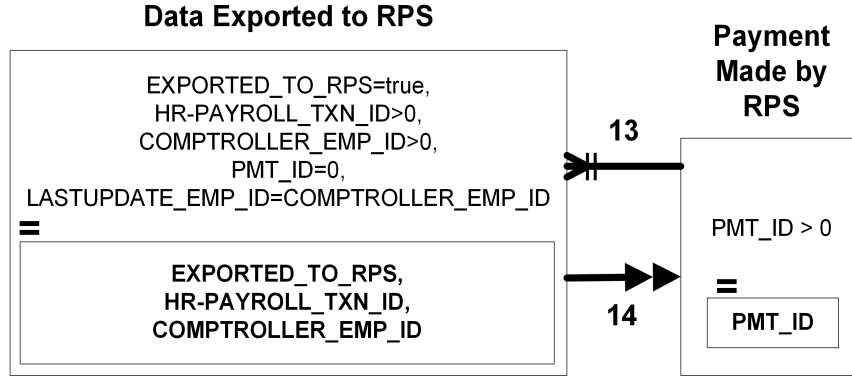


Figure 7.9: DCML canvas consisting only of the *Data Exported to RPS* and *Payment Made by RPS* states. Embedded in this canvas are two constraints pertaining to state transitions (exit and entry restrictions labelled 13 and 14 respectively) and four invariant constraints.

state can be read directly from the diagram:

$$\begin{aligned} & exported_to_rps \wedge hr\text{-}payroll_txn_id > 0 \wedge comptroller_emp_id > 0 \wedge pmt_id = 0 \\ & \wedge lastupdate_emp_id = comptroller_emp_id \end{aligned} \quad (S_2)$$

Similarly, the function defining the PAYMENT MADE BY RPS state can also be read from the canvas as follows:

$$pmt_id > 0 \quad (S_2)$$

A GC179 request is in the DATA EXPORTED INTO RPS or PAYMENT MADE BY RPS states if the above functions (S_2 and S_2 respectively) evaluate to true for the attributes of the artifact. Note that these states are mutually exclusive because of the conditions applicable on the *pmt.id* attribute in their respective functions. Thus, at any point in its history, an artifact can only be present in one of these states. The constraints in this DCML canvas can now be read using the templates provided in Table 3.1 on page 65.

Constraint 13 (exit restriction) is interpreted as $(\bullet S_1 \wedge \neg S_1 \Rightarrow S_2)$ and Constraint 14 (entry restriction) is interpreted as $(S_2 \wedge \neg \bullet S_2 \Rightarrow \bullet S_1)$. There is an invariant constraint on the PAYMENT MADE BY RPS state which can be interpreted in CLTL as $(\bullet pmt_id = pmt_id \text{ since } S_2)$. Similarly there are three invariant constraints on the DATA EXPORTED

INTO RPS state. They are as follows: (i) (\bullet *exported_to_rps = exported_to_rps* **since** S_1), (ii) (\bullet *hr-payroll_txn_id = hr-payroll_txn_id* **since** S_1) and, (iii) (\bullet *comptroller_emp_id = comptroller_emp_id* **since** S_1). The complete canvas constraint system for Figure 7.9 is simply the conjunction of all of the above constraints, and for brevity we omit depicting them in a single conjoined formula.

In a very similar fashion we can construct the artifact constraint system (the complete constraint model for GC179 artifacts) by interpreting the DCML notation and combining individual constraints depicted in the complete DCML diagram (Figure 7.8). This complete constraint model is provided in Appendix A using Lisp. As a final step, the constraint system can be translated into a form that is enforceable by an SQL engine [33, 154, 34].

7.3.4 Verification Results

We were able to generate several (un)satisfiability results to verify the correctness of the behaviour of GC179 artifacts as they evolve in a database. The following statements on the model were formally verified and tested in the Zot model checker, using the CLTL conditions provided in Appendix A:

1. Can an artifact reach the HOURS ACCRUED state while its PAYOUT_REQUESTED attribute is true? In other words, can vacation hours be awarded for a GC179 request for which the employee has requested cash payment?
2. Can a GC179 request for which the employee initially requested a cash payment ever be paid in vacation hours? In other words, can changes to the method of remuneration be made after the request is created?
3. Can the comptroller of records reject the request?
4. Can the request be sent back twice to the employee for changes?
5. Can the request be sent back to the employee for amendments after HR approval (i.e., from the VACATION HOURS CALCULATED or CASH PAY CALCULATED states)?

6. Can a request for payment be exported to the RPS without Section 34 Approval?
7. Can the employee act as the HR coordinator for his/her own GC179 request?

Note that each of the above statements, when tested against the model, has to be interpreted over the attributes of the GC179 artifact in CLTL. As discussed in Chapter 5, this task will be handled by either a database administrator or a policy testing expert who (a) understands the business process associated with the artifact, and (b) can formalize the above statements in constraint temporal logic. We believe that the formal equivalents of the above statements (presented in Appendix A) make it apparent that formulating such properties is not overly challenging. At the same time they provide significant insight on the behaviour of artifacts stored in the database, highlighting situations where an artifact is either over-constrained or under-constrained.

7.3.5 Summary

In Chapter 6 we showed that DCML (and more generally, CLTL) is capable of capturing constraints embedded in a broad class of visual, state oriented descriptions of business processes. Our reasoning was based on the fact that individual workflow patterns can be emulated within LTL constructs. In this section we presented a complete real world business process and its translation into its equivalent in DCML.

Note that DCML is, by design, also a constraint specification language. As a result, the complete specification of a DCML model (e.g., Figure 7.8) can be large, contain many intricately connected states, and perhaps somewhat overwhelming for a single reader. We believe that this is not a shortcoming of the language, but rather an issue of entropy, that is, Figure 7.8 contains substantially more information about the business process than is contained in Figure 7.1. Throughout this thesis we have emphasized the fact that large systems of constraints, whether they are presented visually, in natural language, or in source code, are challenging for unfamiliar readers to understand. Consequently, breaking down even moderately long business processes (such as the one presented in this section) into smaller and more manageable portions is critical to enhance the usability of process

modeling techniques. DCML accomplishes this goal through the use of smaller canvases that integrate easily.

7.4 Will it Work in Practice?

A common criticism of research in the area of policy management is that it fails to answer the following simple question: *How useful is the idea being presented?* Many proposals in this area present a compelling framework and demonstrate usability in a range of scenarios. Some (such as this thesis) will even present case studies based on real life situations to make an argument for the usability of the proposed technique. However, what is typically lacking from such proposals is a comprehensive blueprint using which other researchers can conduct true and fair usability assessments of the proposed policy management system.

In this section we provide quantitative and qualitative guidelines against which usability and applicability of DCML and its associated verification infrastructure can be compared against other techniques. We do so by presenting a general road map of how DCML can be utilized in practice. In addition, we highlight how additional real-world case studies can be conducted and what specific questions would need to be answered to make broad claims about the usability of DCML. We believe that the discussion presented will serve to guide researchers attempting to implement and test the efficacy of our proposed database-oriented policy management system.

The basic requirements to implement and test a policy management system specifically geared towards database systems are as follows:

- *Database*: A schema, data, and the typical query workload are the most basic requirements.
- *Business Processes*: There should be available descriptions of processes and the eminent need to enforce or audit them. Ideally these descriptions should be in a visual notation (similar to workflows), but one may need the co-operation of users in converting textual process description into workflows.

- *Users*: The case study requires domain experts who can serve as corporate policy makers (e.g., lawyers and accountants), policy implementers (users who convert policies into database constraint systems) and policy verification experts or infrastructure managers (e.g., DBA or auditors).

For any case study to be realistic, the above three resources (and specific sub-components) need to be available and accessible. We believe that there will be no major differences among the characteristics of databases that store information about everyday business processes. That is, databases storing records for typical business functions (HR, Sales, etc.) will have similar overall design characteristics. Similarly, since LTL is sufficiently expressive to capture wide ranging workflow semantics, the types and complexity of workflows that various organizations use will also not pose a significant challenge, provided users are able to interpret them for conversion into DCML. However, as elaborated in subsequent sections, the variability in technical skill among different types of users, both within a single organization and across organizations in different industries, will perhaps be the most significant factor in making claims regarding usability.

7.4.1 The Process

The overall challenge of testing usability of any policy management system is that of following the steps toward enforcement described in Chapter 1 and illustrated in Section 7.1. In the specific case of DCML, there are substantial setup and initialization costs that have to be incurred before any rewards can be reaped. For a set of business processes that need to be enforced in the database, the appropriate artifact definitions need to be created and agreed upon by the DBA and policy makers. Then a specific extension or variant of DCML (e.g., ones that may include metric or temporal access control restrictions) need to be agreed upon for various process types. In this stage the DBA will select particular fragments of CLTL and assign meaning to them in the visual state-constraint notation for individual sub-classes of policy makers. Then, policy makers (departmental power users, policy experts for specific areas, or programmers specializing in different areas of compliance) need to be engaged and taught how to create constraint diagrams over artifacts

of interest. These users will then construct visual constraint diagrams over database level artifacts. Finally, a database administrator, armed with intimate knowledge about the business and the database, can begin the task of verifying interactions among multiple constraint systems.

One can argue that such a method introduces several points of error in the interpretation of the specification by involving many different actors. However, a large amount of anecdotal evidence collected during our research suggests that this is unavoidable. Managing a web of ECA rules (essentially similar to a large number of “pre-condition implies post-condition” rules generated by DCML) written by tens of different programmers will also over time become an impossible and expensive proposition. More importantly, the transparency of being able to identify precisely which constraint maps onto which transition in a workflow provides significant reduction in the cost of long term maintenance and evolution of policies.

It is thus best if we analyze both the traditional and our proposed approaches in the context of quantitative and qualitative metrics. The next section provides a summary of the factors that need consideration. The discussion presented is by no means complete but is sufficient to highlight the major questions that require resolution in order to make an assessment of usefulness.

7.4.2 Assessing a Policy Management Framework

The four major categories of cost that we consider relevant are as follows: (1) setup, (2) interpretation of policies, (3) implementation of constraints, and (4) extracting compliance guarantees.

Setup

Most database programmers are familiar with writing triggers, constraints, and assertions. Thus, there is practically zero setup time in the “no-framework” approach and a programmer can begin interpreting a policy document over the schema without delay. However, in

the context of our proposal, artifact definitions have to be agreed upon and various fragments of CLTL have to be made ready in visual notation for different departmental policy makers. Depending on the complexity of the organizational workflows and the database schema, this process can either be time consuming or trivial.

There are some quantifiable metrics during setup that can be identified as relevant indicators of overall cost. For example, the amount of *overlap among artifacts* and the *average number of workflows applicable on a single artifact definition* represent information that could be of interest to future researchers. These indicators are invariably related to the business and how its data storage requirements impact the structural complexity of the database schema. Similarly, the type, structure, and formulation of CLTL constraints preferred by system administrators can also provide insight into how users who may not be familiar with temporal logic respond to the framework. We have to keep in mind that most of the complexity associated with CLTL is hidden to the policy makers through the use of visual notation. However, the DBA or the equivalent supervisory team that decides on a particular fragment of CLTL may itself have to learn about temporal logic and eventually make decisions on *notation*, *types of constraints*, and the *assumptions* under which the framework will be presented and explained to individual policy makers.

Since most quantifiable determinants of cost and complexity are situation-specific (i.e., dependent on policy makers, system administrators, and the complexity of the business), it may be difficult to assess their correlation to the overall setup cost and make generalized statements about their impact. Most notably, individual biases towards certain types of constraints, processes, and workflows in one industry can lead us to make incorrect broad judgements that may not be applicable in other situations. Nonetheless, we hypothesize that if a comprehensive real-life examination of the usability of temporal logics in business situations is conducted, many of these soft indicators of complexity and cost can be identified and discussed in greater detail.

Interpretation

In Chapter 3 we argued that there are many advantages to everyday business users being empowered to express their policies on top of database level artifacts. Intuition suggests

that business users (not necessarily programmers) have intimate knowledge about policies, and if they can directly implement or modify constraints on a database, then significant cost savings can be realized.

To verify our intuition, it is essential to separate the notion of cost during the policy interpretation phase into two components: (a) reduction in incorrect interpretations, and (b) speed or ease of interpretation. In the traditional approach a programmer, if uncertain about the interpretation, must interact with a business user familiar with the given policy for guidance. Thus, the process is not only slower but arguably incurs a higher risk of errors through the involvement of multiple parties. Similarly, we note that high level business policy documents that contain descriptions and specifications of constraints rather than their formal interpretations are best understood by non-programmers. So a business user seems to be best qualified to read policy documents and interpret them in an error-free manner.

Unfortunately, to verify the above hypothesis is far from easy. Putting the “no-framework” approach against an artifact and LTL-constraint-system approach requires that we conduct an analysis of the interpretation capabilities of a programmer against those of a business user. Furthermore, we assume that this business user has now been properly trained in developing constraint systems and that this experiment is conducted in a controlled situation. This assessment is extremely difficult to replicate across several situations. Ideally, we want to be able to make claims about the correctness of an average programmer’s interpretation of a policy document on a database against those made by a business user over a set of pre-defined artifacts and to identify how long the process takes in each case.

Implementation

An examination similar to that presented in the interpretation phase extends into implementation. Once again, we want to pit a programmer against a policy maker. However, as alluded to earlier, at and beyond this phase, the infrastructure of a comprehensive policy modeling framework will likely outperform any manual approach. Practically all metrics ranging from *run-time efficiency of code* to *time taken in development and testing of*

constraints will favour an automated code generation system geared towards a particular database engine.

Nonetheless, it is important to measure this cost so that it can be offset against the initial setup time. Note that, from a cost savings perspective, the programming hours that are saved during implementation should be much higher than the administrative hours during setup. We have to draw out this distinction since different users cost a business different amounts per hour of effort. Thus, an accurate assessment of time saved requires that we measure the time taken by programmers to create triggers for enforcing business rules derived in the previous phase against a time of near-zero in the automated approach.

Maintenance

Maintenance of a set of policies and business rules can be described as the removal of certain constraints and/or the introduction of new ones. The process of maintenance is ongoing and not different from repeating the interpretation and implementation phases. Note, however, that the one-time setup cost incurred under the artifact/LTL-constraint framework now gets further amortized over the lifetime of a business and the database. Consequently, we need to test the hypothesis, first presented in Chapter 3, that the marginal cost to make changes to a policy/constraint system in the “no-framework” method is much higher (and progressively increases with the complexity of the policy set) when compared to the same marginal cost in the artifact/constraint-LTL approach.

Compliance Guarantees and Supporting External Audits

The final aspect in which the usefulness of our proposal needs to be evaluated is in its ability to generate compliance guarantees and to provide some level of assurance to internal and external auditors.

When examining the related problems of governance, risk, and compliance (GRC) in Chapter 2, we observed that it is often difficult to predict what auditors will look for in a given database. Thus, it is imperative to self audit and ensure that there are no inconsistencies within corporate processes. Chapter 5 subsequently revealed that there are

no standardized notions of correctness, inconsistencies, and conflicts when artifact level policies interact. Thus, internal corporate compliance officers in conjunction with DBAs need to rigorously test and subsequently prove that the constraints on a database can never lead to undesirable consequences.

Since we cannot predict what an auditor will be looking for, the question of which properties warrant formal verification is best left for experts in corporate GRC. However, we can focus on the question of how to complete the verification process. Formal verification of hand written code is challenging but still accomplishable. Our proposed system, on the other hand, has significant advantages of automating this process and reducing it to satisfiability and model checking (*k-satisfiability*). However, many questions about artifact constraint systems and verification remain unanswered. For example, how large do we expect these “typical workflows” to be, and how many variables, conditions, and types of functions will be involved in the state definitions. Earlier we had stipulated that these workflows rarely exceed 100 or so states, and analysis of a 1000 states represented an extreme case. Such statements would either be supported or refuted in a longitudinal study.

There are several other issues that were not addressed when evaluating the efficiency of generating proofs in our framework. Foremost is the issue of an upper bound for k when the notion of *k-satisfiability* was introduced in the context of bounded model checking. We alluded to the fact that workflows typically have a maximum “diameter” that essentially represents the longest possible non-looping sequence of state changes in which an artifact can partake. Thus, it would be reasonable to consider that diameter as an upper bound for k . However, without actually having a solid understanding of real-world user-generated constraint systems, it is difficult to directly assess the feasibility of this upper bound. If the time required to model check properties over workflows exceeds user expectations, then certainly a wide variety of issue open up and many possible optimizations and simplifications may have to be examined. Along the same lines are questions pertaining to the interpretation of unsatisfiability results, that is, how best to explain these results to users (e.g., interpreting the least unsatisfiable cores) and the possible fixes for conflicts.

An important metric associated with compliance is that of *policy coverage*. Coverage is a measure that divides the number of policies that a system can implement and reason over

by the total number of policies to which a business is subject. Thus, it is important that we measure the proportion of constraints that can be covered in our framework relative to those that would require a more complex logic for reasoning (e.g., calendar and periodicity constraints). This would provide significant insight into the applicability of a system built around linear temporal logic and possible future extensions.

7.5 Summary and Conclusion

In this chapter we presented a case study highlighting the challenges in converting business processes into a set of database constraints. We then presented a summary of what we believe would be the key areas in which our proposed constraint implementation and management infrastructure can be measured against the existing state of the art. This chapter should have convinced the reader that assessing the viability of any policy and constraint management framework in all settings is extremely difficult. However, we speculate that a longitudinal study on the usability of DCML (or more generally, LTL) in everyday business situations, such as, specification to constraint translation, will reveal many additional concerns that we have not presented in our examination. Having a handle on these “unknown unknowns” will certainly improve our understanding of users’ view of constraints and workflows. Consequently, we believe that taking on this massive challenge of proving usability of a visual-CLTL constraint notation is a significant and worthy avenue of further research in this area.

Chapter 8

Conclusions and Future Work

This chapter concludes this thesis by summarizing the thesis in Section 8.1 and briefly discussing avenues of further research in Section 8.2.

8.1 Summary and Conclusion

This thesis presents a framework for modeling, implementing, and verifying process-oriented constraints in database systems (Figure 8.1). Our work is motivated by the fact that emerging requirements for compliance and auditing will inevitably put relational databases under the microscope of savvy auditors. These auditors are likely to go beyond ensuring that there are no violations of published rules in the current instance, and thus businesses will inevitably need to ensure that their databases will forever remain compliant with a given set of rules, while seamlessly supporting everyday business functions.

This thesis builds on a large amount of infrastructure presented in prior work. However the elegance of our approach lies in its simplicity and the particular way in which the three areas of workflows, temporal integrity constraints, and model checking fit together to solve a problem of extreme significance. We observe that at the surface, databases are collections of arbitrary n-tuples that can evolve in a completely unrestricted fashion. However, it is only when we begin to restrict individual portions of a database based

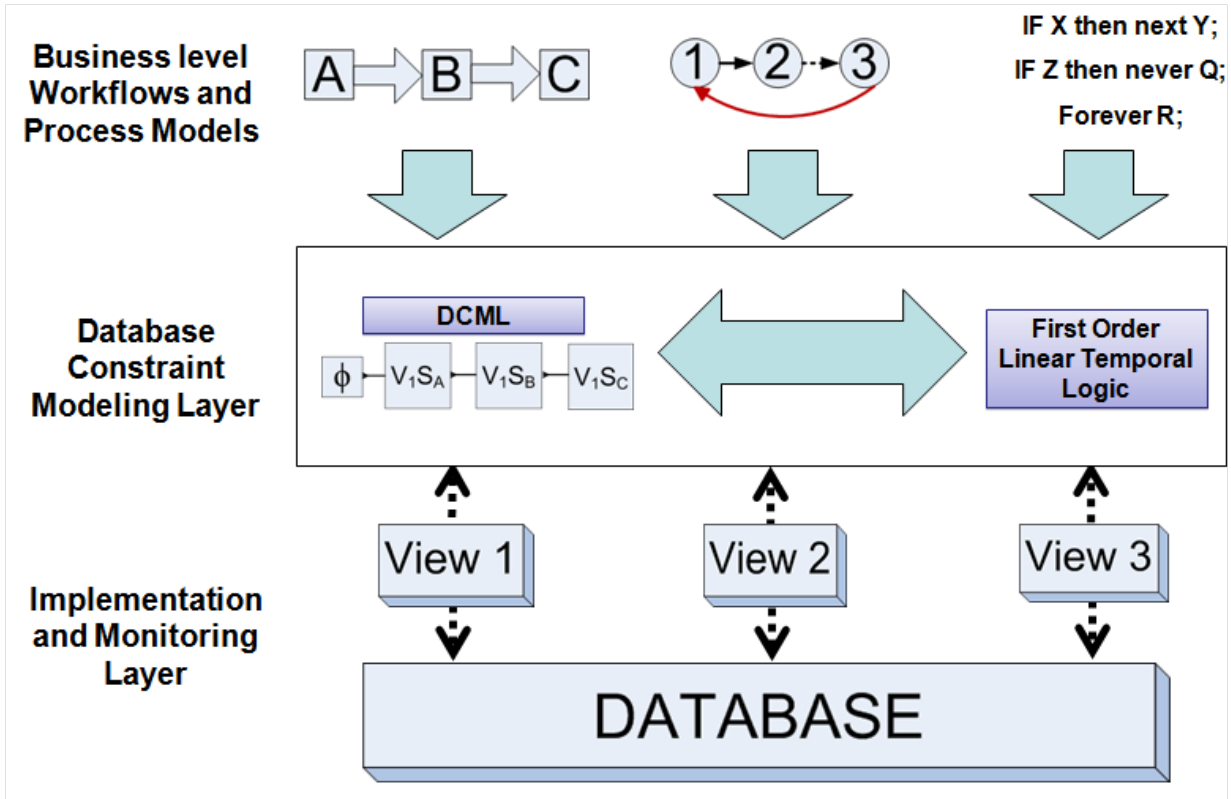


Figure 8.1: A high-level overview of how DCML can be used to implement constraints in database systems. Since business process descriptions exist in varying formats (top of the policy stack), we propose the use of an intermediate constraint specification layer within the database. We denote this as the database constraint modeling layer, in which first order temporal integrity constraints are used to stitch together states derived from the knowledge contained in the process models / workflows. Using DCML, we can then easily implement these constraints and verify a broad range of properties about the interaction of these constraint systems.

on business processes, that a framework of constraints and verification becomes available to answer critical questions. We strongly believe that in the near future there will be increasing need to verify the behaviour of how databases evolve with regards to business rules. The complexity of software systems and their nature of being black-boxes in business environments make the database the key piece of operational evidence on which an auditor can focus. Thus, generating compliance guarantees on the behaviour of the database will be critical in ensuring that a business holds true to its claims of following the published processes.

This work contributes to the state of the art in several different ways. Foremost, it presents an easy to use mechanism that enables the average business user to harness the power of linear temporal logic to visually map out process-centric constraints within a database. Constraints such as process-centric access control were previously well outside the domain of an average user to specify, but our framework makes it easy to implement and maintain constraints in the context of databases. Furthermore, it offers the ability to automatically enforce complex rules, which may have been formulated by non-technical users, as well as allowing a more savvy class of users (internal auditors or DBAs) to verify properties about the interactions of these rule systems. The work is not without its limitations, with a major impediment being the high initial setup cost. However, once a formal framework of artifact definitions and logical-to-visual LTL constraints is agreed upon for a given database, the cost savings achieved over time in modeling, implementing, and verifying complex process-centric constraints will be far greater in magnitude than the initial setup cost.

8.2 Future Work

In Chapter 7 we examined in detail the challenges associated with assessing the usability of a visual-LTL constraint modeling framework. We believe that a longitudinal study on the usability of LTL in business situations represents a significant avenue of future work. In addition, this thesis did not present a complete visual policy modeling software suite for DCML, and we envision that such a tool will help in conducting a case study on usability.

If users are able to graphically draw out DCML models and visualize their own templates of constraints, they may be able to significantly improve their understanding of LTL. The level of insight provided by such a tool will go well beyond assessing the usability of LTL for the users who create DCML models. In fact, it may also provide the much needed insight into the users who will be testing/verifying large integrated constraint systems, and system administrators responsible for auditing and enforcement.

In addition to the above, there are lines of future work within database systems research that we believe to be promising. Our work attempts to connect databases and workflows via state systems that represented integrity constraints specified in LTL. Many new questions arise if we relax the notion of consistency and try to establish connections between workflows and probabilistic state systems such as Bayesian networks and Markov chains. We now provide a brief discussion of these research topics.

8.2.1 Probabilistic Reasoning in Database Level Workflows

Recall that in our infrastructure there is no inherent value or reward associated with compliance. *All constraints are equally important* and failing to meet even a single constraint is deemed unacceptable. However, if we relax this restriction and associate some level of utility or reward with different paths in a business process, we can then introduce the notion of meeting specific goals and targets. In a general model we can assign values to particular paths and state traversals and attempt to restrict or block paths that lead to less than maximum achievable utility.

We can see many situations where the benefit of selectively (perhaps based on a probability function) allowing a transaction that violates a particular “soft-constraint” may outweigh the costs associated with total compliance. As an example, consider an airline business in which each plane has a fixed passenger carrying capacity. The airline has already identified for certain routes the probability distribution for how many passengers do not show up for their flight or cancel at the last minute. In an effort to minimize the number of empty seats for each flight, the airline wishes to selectively violate the constraint that the number of tickets sold must be no greater than the number of seats on the plane, in the hope that the number of no-shows will make the plane run at full capacity. The

airline has developed a model for the risk (cost associated with providing hotel stay, refund, liability, brand image destruction, etc.) for each passenger who does not get boarded, in the event that the number of passengers who show up for their flight exceeds the plane's passenger carrying capacity. How does such a model get encoded as a set of utility-based constraints in a database?

Since a database contains much historical information about the context in which a certain decision is made, the challenge is to encode such models in terms of those decisions. In theory, as a database evolves, so will the parameters of the model, and the decision making process will keep improving to achieve higher total utility.

8.2.2 History Mining and Process Discovery

A database contains a wealth of information pertaining to business processes, and a significant amount of knowledge about these processes is embedded in various object histories. Transactional databases that handle many similar tasks store histories (or artifact oriented event-logs), which can exhibit common patterns that are ready to be exploited in different ways. The idea of automatically creating visual and/or logical descriptions of processes from logs is not new. Processing mining techniques have been examined in great detail in the workflow community, and the reader is directed to the book by van der Aalst [159] as a comprehensive reference. Open source tools, such as ProM [157], are also readily available to aid users in discovering process models from event logs.

However, methods of visualizing processes based on specific needs of database auditors and policy makers is an area that has not received much attention. Databases introduce a significant level of complexity in the problem of event/log mining. For example, we pointed out in Chapter 5 that the history of a given unique row in a table is different from the history of the joins in which it participates. There are several issues pertaining to the interplay of concurrent (possibly related) processes that makes the problem interesting when examined from a database perspective.

Consequently we believe that examining the role of various users and different discovery objectives (auditing, implicit constraint discovery in processes, etc.) in the context of

databases is an excellent topic for further research. Note that compliance generally deals with restricting the state space of a system, whereas process mining attempts to explore and explain this restricted space. However, both can be used in conjunction, for example, to discover variances and interesting behaviour within the restricted state space. Thus, discovery of processes as envisioned in databases can provide us significant insight into how compliance is achieved. It can also reveal avenues of deviation from published processes and lead to additional constraints on a system for even more rigorous compliance.

Appendix A

Zot Model for GC179

Complete Zot Model for the GC179 example presented in Chapter 7

```
(asdf:operate 'asdf:load-op 'ae2zot)
(use-package :trio-utils)

;booleans do not need to be explicitly defined in ZOT
(define-tvar 'GC179_ID *int*)
(define-tvar 'EMP_ID *int*)
(define-tvar 'FILLED_BY_EMP_ID *int*)
;(define-tvar 'CROSS_CHECKED *bool*)
;(define-tvar 'CHANGES_REQUIRED *bool*)
(define-tvar 'APPROVAL_SUPERVISOR_EMP_ID *int*)
(define-tvar 'SEC34_AUTHORITY_EMP_ID *int*)
(define-tvar 'HR_EMP_ID *int*)
(define-tvar 'HOURS_ACCRUED *int*)
(define-tvar 'CASH_PAYMENT_AMOUNT *int*)
(define-tvar 'HR_PAYROLL_TXN_ID *int*)
;(define-tvar 'EXPORTED_TO_RPS *bool*)
;(define-tvar 'PAYOUT_REQUESTED *bool*)
(define-tvar 'COMPTROLLER_EMP_ID *int*)
(define-tvar 'PMT_ID *int*)
(define-tvar 'LASTUPDATE_EMP_ID *int*)

;DEFINITIONS OF STATES
(defvar S.Filled.By.Designate
  (
    &&
    ;State conditions
    ([>] (-V- GC179_ID) 0)
```

```

(>) (-V- EMP_ID) 0)
(>) (-V- FILLED_BY_EMP_ID) 0)
(!! (-P- CROSS_CHECKED))
(!! (-P- CHANGES_REQUIRED))
(=) (-V- APPROVAL_SUPERVISOR_EMP_ID) 0)
(=) (-V- SEC34_AUTHORITY_EMP_ID) 0)
(=) (-V- HR_EMP_ID) 0)
(=) (-V- CASH_PAYMENT_AMOUNT) 0)
(=) (-V- HOURS_ACCRUED) 0)
(=) (-V- HR_PAYROLL_TXN_ID) 0)
(!! (-P- EXPORTED_TO_RPS))
(=) (-V- COMPTROLLER_EMP_ID) 0)
(=) (-V- PMT_ID) 0)
;Access Control
(!=) (-V- FILLED_BY_EMP_ID) (-V- EMP_ID))
(=) (-V- LASTUPDATE_EMP_ID) (-V- FILLED_BY_EMP_ID))
)
)

```

(defvar S_Self_Filled

```

(
  &&
  ;State conditions
  (>) (-V- GC179_ID) 0)
  (>) (-V- EMP_ID) 0)
  (>) (-V- FILLED_BY_EMP_ID) 0)
  (-P- CROSS_CHECKED)
  (!! (-P- CHANGES_REQUIRED))
  (=) (-V- APPROVAL_SUPERVISOR_EMP_ID) 0)
  (=) (-V- SEC34_AUTHORITY_EMP_ID) 0)
  (=) (-V- HR_EMP_ID) 0)
  (=) (-V- CASH_PAYMENT_AMOUNT) 0)
  (=) (-V- HOURS_ACCRUED) 0)
  (=) (-V- HR_PAYROLL_TXN_ID) 0)
  (!! (-P- EXPORTED_TO_RPS))
  (=) (-V- COMPTROLLER_EMP_ID) 0)
  (=) (-V- PMT_ID) 0)
  ;Access Control
  (=) (-V- FILLED_BY_EMP_ID) (-V- EMP_ID))
  (=) (-V- LASTUPDATE_EMP_ID) (-V- EMP_ID))
)
)

```

(defvar S_Cross_Checked

```

(
  &&
  (-P- CROSS_CHECKED)
  (!! (-P- CHANGES_REQUIRED))
)

```



```

        (=[=] (-V- APPROVAL_SUPERVISOR_EMP_ID) 0)
    )
)

(defvar S.Changes_Required
  (
    &&
    (!! (-P- CROSS_CHECKED)) ;*****
    (-P- CHANGES_REQUIRED);*****
    (=[=] (-V- APPROVAL_SUPERVISOR_EMP_ID) 0)
    (=[=] (-V- SEC34_AUTHORITY_EMP_ID) 0)
    (=[=] (-V- HR_EMP_ID) 0)
    (=[=] (-V- CASH_PAYMENT_AMOUNT) 0)
    (=[=] (-V- HOURS_ACCRUED) 0)
    (=[=] (-V- HR-PAYROLL_TXN_ID) 0)
    (!! (-P- EXPORTED_TO_RPS))
    (=[=] (-V- COMPROLLER_EMP_ID) 0)
    (=[=] (-V- PMT_ID) 0)
  )
)

(defvar S.Approved_By_Supervisor
  (
    &&
    ([>] (-V- APPROVAL_SUPERVISOR_EMP_ID) 0);*****
    (=[=] (-V- SEC34_AUTHORITY_EMP_ID) 0);*****
    ;Access Control
    (=[=] (-V- LASTUPDATE_EMP_ID) (-V- APPROVAL_SUPERVISOR_EMP_ID))
    (![!=] (-V- EMP_ID) (-V- APPROVAL_SUPERVISOR_EMP_ID))
  )
)

(defvar S.Approved_By_Sect34_Authority
  (
    &&
    ([>] (-V- SEC34_AUTHORITY_EMP_ID) 0);*****
    (=[=] (-V- HR_EMP_ID) 0)
    ;Access Control
    (=[=] (-V- LASTUPDATE_EMP_ID) (-V- SEC34_AUTHORITY_EMP_ID))
    (![!=] (-V- EMP_ID) (-V- SEC34_AUTHORITY_EMP_ID))
    (![!=] (-V- APPROVAL_SUPERVISOR_EMP_ID) (-V- SEC34_AUTHORITY_EMP_ID))
  )
)

(defvar S.Vacation_Hours_Calculated
  (
    &&

```

```

([>] (-V- HR_EMP_ID) 0);*****
([=] (-V- CASH_PAYMENT_AMOUNT) 0)
([>] (-V- HOURS_ACCRUED) 0)
(!! (-P- PAYOUT_REQUESTED))
(!! (-P- EXPORTED_TO_RPS))
([=] (-V- PMT_ID) 0)
([=] (-V- COMPTROLLER_EMP_ID) 0)
;Access Control
([=] (-V- LASTUPDATE_EMP_ID) (-V- HR_EMP_ID))
)
)

(defvar S.Cash_Pay_Calculated
(
  &&
  ([>] (-V- HR_EMP_ID) 0);*****
  ([>] (-V- CASH_PAYMENT_AMOUNT) 0)
  ([=] (-V- HOURS_ACCRUED) 0)
  (-P- PAYOUT_REQUESTED)
  (!! (-P- EXPORTED_TO_RPS))
  ([=] (-V- PMT_ID) 0)
  ([=] (-V- COMPTROLLER_EMP_ID) 0)
  ;Access Control
  ([=] (-V- LASTUPDATE_EMP_ID) (-V- HR_EMP_ID))
)
)

(defvar S.Hours_Accrued
(
  &&
  (!! (-P- EXPORTED_TO_RPS))
  ([>] (-V- HR-PAYROLL_TXN_ID) 0)
  ([>] (-V- COMPTROLLER_EMP_ID) 0)
  ;Access Control
  ([=] (-V- LASTUPDATE_EMP_ID) (-V- COMPTROLLER_EMP_ID))
)
)

(defvar S.EXPORTED_TO_RPS
(
  &&
  (-P- EXPORTED_TO_RPS)
  ([>] (-V- HR-PAYROLL_TXN_ID) 0)
  ([>] (-V- COMPTROLLER_EMP_ID) 0)
  ([=] (-V- PMT_ID) 0)
  ;Access Control
  ([=] (-V- LASTUPDATE_EMP_ID) (-V- COMPTROLLER_EMP_ID))
)
)

```

```

    )
)

(defvar S.PAYMENT_MADE
  (
    &&
    ([>] (-V- PMT.ID) 0)
  )
)

(defvar S.Rejected
  (
    &&
    (-P- PHI)
  )
)

(defvar model_guards_and_invariants
  (
    &&
    (;S_Self_Filled
      alwf
      (
        ->
          S_Self_Filled
          (
            &&
            ;Future Invariants
            (alwf ([=] (-V- GC179.ID) (futr (-V- GC179.ID) 1)))
            (alwf ([=] (-V- EMP.ID) (futr (-V- EMP.ID) 1)))
            (alwf ([=] (-V- FILLED.BY.EMP.ID) (futr (-V- FILLED.BY.EMP.ID) 1)))
          )
        )
      )
    (;S_Cross_Checked
      alwf
      (
        ->
          S_Cross_Checked
          (
            &&
            ;Guarded Variables
            ([=] (-V- SEC34.AUTHORITY.EMP.ID) (past (-V- SEC34.AUTHORITY.EMP.ID) 1))
            ([=] (-V- HR.EMP.ID) (past (-V- HR.EMP.ID) 1))
            ([=] (-V- CASH.PAYMENT.AMOUNT) (past (-V- CASH.PAYMENT.AMOUNT) 1))
          )
        )
      )
  )
)

```

```

)
([=] (-V- HOURS_ACCRUED) (past (-V- HOURS_ACCRUED) 1))
([=] (-V- HR-PAYROLL_TXN_ID) (past (-V- HR-PAYROLL_TXN_ID) 1))
(|| (&& (-P- EXPORTED_TO_RPS) (past (-P- EXPORTED_TO_RPS) 1)) (&& (!! (-
P- EXPORTED_TO_RPS)) (!! (past (-P- EXPORTED_TO_RPS) 1))))
(|| (&& (-P- PAYOUT_REQUESTED) (past (-P- PAYOUT_REQUESTED) 1)) (&&
 (!! (-P- PAYOUT_REQUESTED)) (!! (past (-P- PAYOUT_REQUESTED) 1))))
([=] (-V- COMPTROLLER_EMP_ID) (past (-V- COMPTROLLER_EMP_ID) 1))
([=] (-V- PMT_ID) (past (-V- PMT_ID) 1))
)
)
)
(;S_Filled_By_Designate
  alwf
  (
  ->
    S_Filled_By_Designate
    (
    &&
    ;Future Invariants
    (alwf ([=] (-V- GC179_ID) (futr (-V- GC179_ID) 1)))
    (alwf ([=] (-V- EMP_ID) (futr (-V- EMP_ID) 1)))
    (alwf ([=] (-V- FILLED_BY_EMP_ID) (futr (-V- FILLED_BY_EMP_ID) 1)))
    )
  )
)
(;S_Approved_By_Supervisor
  alwf
  (
  ->
    S_Approved_By_Supervisor
    (
    &&
    ;Guarded Variables
    (|| (&& (-P- CROSS_CHECKED) (past (-P- CROSS_CHECKED) 1)) (&& (!! (-P-
CROSS_CHECKED)) (!! (past (-P- CROSS_CHECKED) 1))))
    (|| (&& (-P- CHANGES_REQUIRED) (past (-P- CHANGES_REQUIRED) 1)) (&&
 (!! (-P- CHANGES_REQUIRED)) (!! (past (-P- CHANGES_REQUIRED) 1))))
    ([=] (-V- SEC34_AUTHORITY_EMP_ID) (past (-V- SEC34_AUTHORITY_EMP_ID)
1))
    ([=] (-V- HR_EMP_ID) (past (-V- HR_EMP_ID) 1))
    ([=] (-V- CASH_PAYMENT_AMOUNT) (past (-V- CASH_PAYMENT_AMOUNT) 1))
    )
    ([=] (-V- HOURS_ACCRUED) (past (-V- HOURS_ACCRUED) 1))
    ([=] (-V- HR-PAYROLL_TXN_ID) (past (-V- HR-PAYROLL_TXN_ID) 1))
  )
)

```

```

(|&& (-P- EXPORTED_TO_RPS) (past (-P- EXPORTED_TO_RPS) 1)) (&& (!! (-
P- EXPORTED_TO_RPS)) (!! (past (-P- EXPORTED_TO_RPS) 1))))
(|&& (-P- PAYOUT_REQUESTED) (past (-P- PAYOUT_REQUESTED) 1)) (&&
 (!! (-P- PAYOUT_REQUESTED)) (!! (past (-P- PAYOUT_REQUESTED) 1))))
(=[] (-V- COMPTROLLER_EMP_ID) (past (-V- COMPTROLLER_EMP_ID) 1))
(=[] (-V- PMT_ID) (past (-V- PMT_ID) 1))
)
)
)

```

(;S-Approved_By_Sect34-Authority

```

alwf
(
->
S.Approved_By_Sect34-Authority
(
&&
;Guarded Variables
(|&& (-P- CROSS_CHECKED) (past (-P- CROSS_CHECKED) 1)) (&& (!! (-P-
CROSS_CHECKED)) (!! (past (-P- CROSS_CHECKED) 1))))
(|&& (-P- CHANGES_REQUIRED) (past (-P- CHANGES_REQUIRED) 1)) (&&
 (!! (-P- CHANGES_REQUIRED)) (!! (past (-P- CHANGES_REQUIRED) 1))))
(=[] (-V- APPROVAL_SUPERVISOR_EMP_ID) (past (-V-
APPROVAL_SUPERVISOR_EMP_ID) 1))
(=[] (-V- HR_EMP_ID) (past (-V- HR_EMP_ID) 1))
(=[] (-V- CASH_PAYMENT_AMOUNT) (past (-V- CASH_PAYMENT_AMOUNT) 1))
)
(=[] (-V- HOURS_ACCRUED) (past (-V- HOURS_ACCRUED) 1))
(=[] (-V- HR-PAYROLL_TXN_ID) (past (-V- HR-PAYROLL_TXN_ID) 1))
(|&& (-P- EXPORTED_TO_RPS) (past (-P- EXPORTED_TO_RPS) 1)) (&& (!! (-
P- EXPORTED_TO_RPS)) (!! (past (-P- EXPORTED_TO_RPS) 1))))
(|&& (-P- PAYOUT_REQUESTED) (past (-P- PAYOUT_REQUESTED) 1)) (&&
 (!! (-P- PAYOUT_REQUESTED)) (!! (past (-P- PAYOUT_REQUESTED) 1))))
(=[] (-V- COMPTROLLER_EMP_ID) (past (-V- COMPTROLLER_EMP_ID) 1))
(=[] (-V- PMT_ID) (past (-V- PMT_ID) 1))
)
)
)

```

(;S-Vacation_Hours_Calculated

```

alwf
(
->
S.Vacation_Hours_Calculated
(

```

```

&&
;Guarded Variables
(|| (&& (-P- CROSS_CHECKED) (past (-P- CROSS_CHECKED) 1)) (&& (!! (-P-
  CROSS_CHECKED)) (!! (past (-P- CROSS_CHECKED) 1))))
(|| (&& (-P- CHANGES_REQUIRED) (past (-P- CHANGES_REQUIRED) 1)) (&&
  (!! (-P- CHANGES_REQUIRED)) (!! (past (-P- CHANGES_REQUIRED) 1))))
([=] (-V- APPROVAL_SUPERVISOR_EMP_ID) (past (-V-
  APPROVAL_SUPERVISOR_EMP_ID) 1))
([=] (-V- SEC34_AUTHORITY_EMP_ID) (past (-V- SEC34_AUTHORITY_EMP_ID)
  1))
([=] (-V- HR-PAYROLL_TXN_ID) (past (-V- HR-PAYROLL_TXN_ID) 1))
(|| (&& (-P- PAYOUT_REQUESTED) (past (-P- PAYOUT_REQUESTED) 1)) (&&
  (!! (-P- PAYOUT_REQUESTED)) (!! (past (-P- PAYOUT_REQUESTED) 1))))
(|| (&& (-P- EXPORTED_TO_RPS) (past (-P- EXPORTED_TO_RPS) 1)) (&& (!! (-P-
  EXPORTED_TO_RPS)) (!! (past (-P- EXPORTED_TO_RPS) 1))))
([=] (-V- COMPTROLLER_EMP_ID) (past (-V- COMPTROLLER_EMP_ID) 1))
([=] (-V- PMT_ID) (past (-V- PMT_ID) 1))
;Future Invariants
(alwf (|| (&& (-P- CROSS_CHECKED) (futr (-P- CROSS_CHECKED) 1)) (&& (!! (-P-
  CROSS_CHECKED)) (!! (futr (-P- CROSS_CHECKED) 1))))
(alwf (|| (&& (-P- CHANGES_REQUIRED) (futr (-P- CHANGES_REQUIRED) 1))
  (&& (!! (-P- CHANGES_REQUIRED)) (!! (futr (-P- CHANGES_REQUIRED)
  1))))))
(alwf ([=] (-V- APPROVAL_SUPERVISOR_EMP_ID) (futr (-V-
  APPROVAL_SUPERVISOR_EMP_ID) 1)))
(alwf ([=] (-V- SEC34_AUTHORITY_EMP_ID) (futr (-V-
  SEC34_AUTHORITY_EMP_ID) 1)))
(alwf (|| (&& (-P- PAYOUT_REQUESTED) (futr (-P- PAYOUT_REQUESTED) 1))
  (&& (!! (-P- PAYOUT_REQUESTED)) (!! (futr (-P- PAYOUT_REQUESTED)
  1))))))
(alwf ([=] (-V- CASH_PAYMENT_AMOUNT) (futr (-V-
  CASH_PAYMENT_AMOUNT) 1)))
(alwf ([=] (-V- PMT_ID) (futr (-V- PMT_ID) 1)))
)
)
)

(;S_Cash_Pay_Calculated
  alwf
  (
  ->
    S_Cash_Pay_Calculated
    (
    &&
    ;Guarded Variables
    (|| (&& (-P- CROSS_CHECKED) (past (-P- CROSS_CHECKED) 1)) (&& (!! (-P-
      CROSS_CHECKED)) (!! (past (-P- CROSS_CHECKED) 1))))
    (|| (&& (-P- CHANGES_REQUIRED) (past (-P- CHANGES_REQUIRED) 1)) (&&

```

```

        (!! (-P- CHANGES_REQUIRED)) (!! (past (-P- CHANGES_REQUIRED) 1))))
([=] (-V- APPROVAL_SUPERVISOR_EMP_ID) (past (-V-
        APPROVAL_SUPERVISOR_EMP_ID) 1))
([=] (-V- SEC34_AUTHORITY_EMP_ID) (past (-V- SEC34_AUTHORITY_EMP_ID
        1))
([=] (-V- HR-PAYROLL_TXN_ID) (past (-V- HR-PAYROLL_TXN_ID) 1))
(|| (&& (-P- PAYOUT_REQUESTED) (past (-P- PAYOUT_REQUESTED) 1)) (&&
        (!! (-P- PAYOUT_REQUESTED)) (!! (past (-P- PAYOUT_REQUESTED) 1))))
(|| (&& (-P- EXPORTED_TO_RPS) (past (-P- EXPORTED_TO_RPS) 1)) (&& (!! (-
        P- EXPORTED_TO_RPS)) (!! (past (-P- EXPORTED_TO_RPS) 1))))
([=] (-V- COMPTROLLER_EMP_ID) (past (-V- COMPTROLLER_EMP_ID) 1))
([=] (-V- PMT_ID) (past (-V- PMT_ID) 1))
;Future Invariants
(alwf (|| (&& (-P- CROSS_CHECKED) (futr (-P- CROSS_CHECKED) 1)) (&& (!! (-
        P- CROSS_CHECKED)) (!! (futr (-P- CROSS_CHECKED) 1))))))
(alwf (|| (&& (-P- CHANGES_REQUIRED) (futr (-P- CHANGES_REQUIRED) 1))
        (&& (!! (-P- CHANGES_REQUIRED)) (!! (futr (-P- CHANGES_REQUIRED)
        1))))))
(alwf ([=] (-V- APPROVAL_SUPERVISOR_EMP_ID) (futr (-V-
        APPROVAL_SUPERVISOR_EMP_ID) 1)))
(alwf ([=] (-V- SEC34_AUTHORITY_EMP_ID) (futr (-V-
        SEC34_AUTHORITY_EMP_ID) 1)))
(alwf (|| (&& (-P- PAYOUT_REQUESTED) (futr (-P- PAYOUT_REQUESTED) 1))
        (&& (!! (-P- PAYOUT_REQUESTED)) (!! (futr (-P- PAYOUT_REQUESTED)
        1))))))
(alwf ([=] (-V- HOURS_ACCRUED) (futr (-V- HOURS_ACCRUED) 1)))
)
)
)

(S_Hours_Accrued
  alwf
  (
  ->
    S_Hours_Accrued
    (
    &&
    ;Future Invariants
    (alwf (|| (&& (-P- EXPORTED_TO_RPS) (futr (-P- EXPORTED_TO_RPS) 1)) (&&
            (!! (-P- EXPORTED_TO_RPS)) (!! (futr (-P- EXPORTED_TO_RPS) 1))))))
    (alwf ([=] (-V- HR-PAYROLL_TXN_ID) (futr (-V- HR-PAYROLL_TXN_ID) 1)))
    (alwf ([=] (-V- COMPTROLLER_EMP_ID) (futr (-V- COMPTROLLER_EMP_ID) 1)
            ))
    )
  )
)

(S_EXPORTED_TO_RPS

```

```

    alwf
    (
    ->
        S_EXPORTED_TO_RPS
        (
        &&
        ;Future Invariants
        (alwf (|| (&& (-P- EXPORTED_TO_RPS) (futr (-P- EXPORTED_TO_RPS) 1)) (&&
            (!! (-P- EXPORTED_TO_RPS)) (!! (futr (-P- EXPORTED_TO_RPS) 1))))))
        (alwf ([=] (-V- HR-PAYROLL_TXN_ID) (futr (-V- HR-PAYROLL_TXN_ID) 1)))
        (alwf ([=] (-V- COMPTROLLER_EMP_ID) (futr (-V- COMPTROLLER_EMP_ID) 1)
            ))
        )
    )
)

( ;S_PAYMENT_MADE
    alwf
    (
    ->
        S_PAYMENT_MADE
        (
        &&
        ;Future Invariants
        (alwf ([=] (-V- PMT.ID) (futr (-V- PMT.ID) 1)))
        )
    )
)
)

```

```

(defvar Instantiation_Constraints
  ( &&
    (!! (-P- PHI)) ;Lifecycle not terminated and
    (|| S_Self_Filled S_Filled_By_Designate); Must be in at least one of the starting states
  )
)

```

```

(defvar Termination_Constraints
  (&&
    ;Previously PHI implies currently PHI
    (alw ;Always or Globally
      (
      ->
        (past (-P- PHI) 1) ;Previously PHI
        (-P- PHI); Currently PHI
      )
    )
  )
)

```



```

)
)

;PHI implies all attributes retain previous values
(alw
(
->
(-P- PHI)
(&&
[=] (-V- GC179_ID) (past (-V- GC179_ID) 1))
[=] (-V- EMP_ID) (past (-V- EMP_ID) 1))
[=] (-V- FILLED_BY_EMP_ID) (past (-V- FILLED_BY_EMP_ID) 1))
(|| (&& (-P- CROSS_CHECKED) (past (-P- CROSS_CHECKED) 1)) (&& (! (-P- CROSS_CHECKED)) (!
(past (-P- CROSS_CHECKED) 1))))
(|| (&& (-P- CHANGES_REQUIRED) (past (-P- CHANGES_REQUIRED) 1)) (&& (! (-P-
CHANGES_REQUIRED)) (! (past (-P- CHANGES_REQUIRED) 1))))
[=] (-V- APPROVAL_SUPERVISOR_EMP_ID) (past (-V- APPROVAL_SUPERVISOR_EMP_ID) 1))
[=] (-V- SEC34_AUTHORITY_EMP_ID) (past (-V- SEC34_AUTHORITY_EMP_ID) 1))
[=] (-V- HR_EMP_ID) (past (-V- HR_EMP_ID) 1))
[=] (-V- HR_PAYROLL_TXN_ID) (past (-V- HR_PAYROLL_TXN_ID) 1))
[=] (-V- HOURS_ACCRUED) (past (-V- HOURS_ACCRUED) 1))
[=] (-V- CASH_PAYMENT_AMOUNT) (past (-V- CASH_PAYMENT_AMOUNT) 1))
(|| (&& (-P- PAYOUT_REQUESTED) (past (-P- PAYOUT_REQUESTED) 1)) (&& (! (-P-
PAYOUT_REQUESTED)) (! (past (-P- PAYOUT_REQUESTED) 1))))
(|| (&& (-P- EXPORTED_TO_RPS) (past (-P- EXPORTED_TO_RPS) 1)) (&& (! (-P-
EXPORTED_TO_RPS)) (! (past (-P- EXPORTED_TO_RPS) 1))))
[=] (-V- COMPTROLLER_EMP_ID) (past (-V- COMPTROLLER_EMP_ID) 1))
[=] (-V- PMT_ID) (past (-V- PMT_ID) 1))
[=] (-V- LASTUPDATE_EMP_ID) (past (-V- LASTUPDATE_EMP_ID) 1))
)
)
)
)
)
)
)

```

```

(defvar modelconstraints
(
&&

    (; Constraint 2
        alwf
        (
        ->
            (
            &&
                (past S-Filled.By-Designate 1)
            )
        )
    )
)
)

```

```

                (!! S_Filled_By_Designate)
            )
            (!! S_Cross_Checked S_Changes_Required S_Rejected)
        )
    )
    (; Constraint 3
      alwf
      (
      ->
        (
          &&
            (past S_Changes_Required 1)
            (!! S_Changes_Required)
          )
          (!! S_Filled_By_Designate S_Self_Filled S_Rejected)
        )
      )
    )
    (; Constraint 4
      alwf
      (
      ->
        (
          &&
            (!! (past S_Approved_By_Supervisor 1))
            S_Approved_By_Supervisor
          )
          (past S_Cross_Checked 1)
        )
      )
    )
    (; Constraint 5
      alwf
      (
      ->
        (
          &&
            (past S_Approved_By_Supervisor 1)
            (!! S_Approved_By_Supervisor)
          )
          (!! S_Approved_By_Sect34_Authority S_Rejected S_Changes_Required)
        )
      )
    )
    (; Constraint 6
      alwf
      (

```

```

->
  (
    &&
      (!! (past S_Approved_By_Sect34_Authority 1))
      S_Approved_By_Sect34_Authority
    )
  (past S_Approved_By_Supervisor 1)
)
)

(; Constraint 7
  alwf
  (
    ->
      (
        &&
          (past S_Approved_By_Sect34_Authority 1)
          (!! S_Approved_By_Sect34_Authority)
        )
      (|| S_Vacation_Hours_Calculated S_Cash_Pay_Calculated S_Rejected S_Changes_Required)
    )
  )
)

(; Constraint 8
  alwf
  (
    ->
      (
        &&
          (!! (past S_Vacation_Hours_Calculated 1))
          S_Vacation_Hours_Calculated
        )
      (past S_Approved_By_Sect34_Authority 1)
    )
  )
)

(; Constraint 9
  alwf
  (
    ->
      (
        &&
          (!! (past S_Cash_Pay_Calculated 1))
          S_Cash_Pay_Calculated
        )
      (past S_Approved_By_Sect34_Authority 1)
    )
  )
)

```

```

(; Constraint 10
  alwf
  (
    ->
      (
        &&
          (!! (past S_Hours_Accrued 1))
          S_Hours_Accrued
        )
      (past S_Vacation_Hours_Calculated 1)
    )
  )

(; Constraint 11
  alwf
  (
    ->
      (
        &&
          (past S_Vacation_Hours_Calculated 1)
          (!! S_Vacation_Hours_Calculated)
        )
      (|| S_Hours_Accrued)
    )
  )

(; Constraint 12
  alwf
  (
    ->
      (
        &&
          (!! (past S_EXPORTED_TO_RPS 1))
          S_EXPORTED_TO_RPS
        )
      (past S_Cash_Pay_Calculated 1)
    )
  )

(; Constraint 13
  alwf
  (
    ->
      (
        &&
          (past S_Cash_Pay_Calculated 1)
          (!! S_Cash_Pay_Calculated)
        )
      )
  )

```

```

        (|| S_EXPORTED_TO_RPS)
    )
)

(; Constraint 13
  alwf
  (
    ->
    (
      &&
      (
        (!! (past S_PAYMENT_MADE 1))
        S_PAYMENT_MADE
      )
      (past S_EXPORTED_TO_RPS 1)
    )
  )
)

(; Constraint 14
  alwf
  (
    ->
    (
      &&
      (
        (past S_EXPORTED_TO_RPS 1)
        (!! S_EXPORTED_TO_RPS)
      )
      (|| S_PAYMENT_MADE)
    )
  )
)
)

)

(ae2zot:zot 10
  (
    &&
    Instantiation_Constraints
    Termination_Constraints
    modelconstraints
    model_guards_and_invariants
    ;Check conditions described in Section 7.3.4
    ;(somf (ℒℒ S_Hours_Accrued (-P- PAYOUT_REQUESTED) )); -- Check 1
    ;(somf (ℒℒ (-P- PAYOUT_REQUESTED) (somf S_Hours_Accrued) )); -- Check 2
    ;(somf (ℒℒ S_Cash_Pay_Calculated (futr (-P- PHI) 1) )); -- Check 3
    ;(somf (ℒℒ S_Changes_Required (somf (ℒℒ (!! S_Changes_Required) (somf S_Changes_Required)) )); --
      Check 4
    ;(|| (somf (ℒℒ S_Cash_Pay_Calculated (somf S_Changes_Required)) ) (somf (ℒℒ
      S_Vacation_Hours_Calculated (somf S_Changes_Required)) )); -- Check 5
  )
)

```

```
;(sopf (SOP S_PAYMENT_MADE ([=] (-V- APPROVAL_SUPERVISOR_EMP_ID) 0))) ;-- Check 6
;(sopf (SOP ([=] (-V- HR_EMP_ID) (-V- EMP_ID)) ([>] (-V- HR_EMP_ID) 0)))-- Check 7
)
:logic :QF_UFLIA
)
```

References

- [1] 104th Congress of the United States of America. The Health Insurance Portability and Accountability Act of 1996 (HIPAA) - Privacy and Security Rules.
- [2] 107th Congress of the United States of America. Sarbanes-Oxley Act of 2002.
- [3] Martin Abadi. Logic in Access Control. In *18th Annual IEEE Symposium on Logic in Computer Science, 2003*, pages 228–233.
- [4] Mark S. Ackerman, Lorrie Faith Cranor, and Joseph Reagle. Privacy in e-commerce: Examining user scenarios and privacy preferences. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 1–8. ACM, 1999.
- [5] Atul Adya, José A. Blakeley, Sergey Melnik, and S. Muralidhar. Anatomy of the ADO.NET entity framework. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07*, pages 877–888, New York, NY, USA, 2007. ACM.
- [6] Agriculture and Agri-Food Canada. Audit of Management Practices Extra Duty Pay, http://www5.agr.gc.ca/resources/prod/doc/info/audit-exam/pdf/ex_dut_e.pdf.
- [7] Ruth Sara Aguilar-Saven. Business Process Modelling: Review and Framework. *International Journal of Production Economics*, 90(2):129–149, 2004.
- [8] Rajeev Alur and Thomas A. Henzinger. Logics and Models of Real Time: A Survey. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 74–106, London, UK, UK, 1992. Springer-Verlag.

- [9] Rajeev Alur, Ken Mcmillan, and Doron Peled. Model-Checking of Correctness Conditions for Concurrent Objects. In *Information and Computation*, pages 219–228. IEEE, 1996.
- [10] Anne Anderson. A comparison of two privacy policy languages: EPAL and XACML. In *Proceedings of the 3rd ACM workshop on Secure Web Services*, pages 53–60. ACM Press, 2006.
- [11] Annie I. Anton, Julia B. Eart, Matthew W. Vail, Neha Jain, Carrie M. Gheen, and Jack M. Frink. HIPAA’s Effect on Web Site Privacy Policies. *IEEE Security and Privacy*, 5(1):45–52, January 2007.
- [12] Alessandro Armando and Serena Elisa Ponta. Model checking of security-sensitive business processes. In *Proceedings of the 6th International Conference on Formal Aspects in Security and Trust, FAST’09*, pages 66–80, Berlin, Heidelberg, 2010. Springer-Verlag.
- [13] Paul Ashley, Satoshi Hada, Günter Karjoth, Calvin Powers, and Matthias Schunter. Enterprise Privacy Authorization Language (EPAL 1.2). Technical report, IBM, 2003.
- [14] Ahmed A. Ataullah, Ashraf Aboulnaga, and Frank Wm. Tompa. Records retention in relational database systems. In *Proceedings of the 17th ACM conference on Information and Knowledge Management, CIKM ’08*, pages 873–882, New York, NY, USA, 2008. ACM.
- [15] Ahmed Ayaz Ataullah. A Framework for Records Management in Relational Database Systems, Masters Thesis at the University of Waterloo, Ontario, Canada. 2008.
- [16] Mohammad Badawy and Karel Richta. Deriving Triggers from UML/OCL Specification. In *Information Systems Development, ISBN: 9781461349501*, pages 305–315. Springer US, 2002.

- [17] Philippe Balbiani and Jean-François Condotta. Computational Complexity of Propositional Linear Temporal Logics Based on Qualitative Spatial or Temporal Reasoning. In *Proceedings of the 4th International Workshop on Frontiers of Combining Systems, FroCoS '02*, pages 162–176, London, UK, UK, 2002. Springer-Verlag.
- [18] David Basin, Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, pages 23–34. ACM, 2010.
- [19] David Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In *Computer Aided Verification*, pages 1–18. Springer, 2010.
- [20] David Baumer, Julia Brande Earp, and Fay Cobb Payton. Privacy of medical records: IT implications of HIPAA. *ACM SIGCAS Computers and Society*, 30(4):40–47, December 2000.
- [21] Jörg Becker, Michael Rosemann, and Christoph Uthmann. Guidelines of business process modeling. In *Business Process Management*, volume 1806 of *Lecture Notes in Computer Science*, pages 30–49. Springer Berlin Heidelberg, 2000.
- [22] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: Model-checking techniques and tools, ISBN:3540415238*. Springer Publishing Company, Incorporated, 2010.
- [23] Philip Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of data*, pages 1–12. ACM, 2007.
- [24] Marcello M Bersani, Achille Frigeri, Angelo Morzenti, Matteo Pradella, Matteo Rossi, and Pierluigi San Pietro. Constraint LTL Satisfiability Checking without Automata. *arXiv preprint arXiv:1205.0946*, 2012.
- [25] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):191–233, 2001.

- [26] Jeroen Blox. BPMN 2 BPEL: Research on mapping BPMN to BPEL. Master's Thesis at Eindhoven University of Technology, Netherlands, 2009.
- [27] Travis Breaux and Annie Antón. Analyzing regulatory rules for privacy and security requirements. *IEEE Transactions on Software Engineering*, 34(1):5–20, 2008.
- [28] Travis D. Breaux, Matthew W. Vail, and Annie I. Anton. Towards regulatory compliance: Extracting rights and obligations to align requirements with regulations. In *Proceedings of the 14th IEEE International Requirements Engineering Conference, RE '06*, pages 46–55, Washington, DC, USA, 2006. IEEE Computer Society.
- [29] Carolyn A. Brodie, Clare-Marie Karat, and John Karat. An empirical study of natural language parsing of privacy policy rules using the SPARCLE policy workbench. In *Proceedings of the second Symposium on Usable Privacy and Security, SOUPS '06*, pages 8–19, New York, NY, USA, 2006. ACM.
- [30] Stefano Ceri, Roberta Cochrane, and Jennifer Widom. Practical applications of triggers and constraints: Success and lingering issues (10-year award). In *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00*, pages 254–262, San Francisco, CA, USA, 2000.
- [31] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems (TODS)*, 19(3):367–422, 1994.
- [32] Young B. Choi, Kathleen E. Capitan, Joshua S. Krause, and Meredith M. Streeper. Challenges Associated with Privacy in Health Care Industry: Implementation of HIPAA and the Security Rules. *Journal of Medical Systems*, 30(1):57–64, February 2006.
- [33] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems (TODS)*, 20(2):149–186, 1995.

- [34] Jan Chomicki and David Toman. Implementing Temporal Integrity Constraints Using an Active DBMS. *IEEE Transactions on Knowledge and Data Engineering*, 7:566–582, August 1995.
- [35] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification, CAV '02*, pages 359–364, London, UK, UK, 2002. Springer-Verlag.
- [36] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
- [37] Edmund M. Clarke, David E Long, and Kenneth L McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 353–362, 1989.
- [38] Roberta Cochrane, Hamid Pirahesh, and Nelson Mendonça Mattos. Integrating Triggers and Declarative Constraints in SQL Database Systems. In *Proceedings of the 22th International Conference on Very Large Data Bases, VLDB '96*, pages 567–578, San Francisco, CA, USA, 1996.
- [39] Juri L. De Coi, Philipp Krger, Daniel Olmedilla, and Sergej Zerr. Using Natural Language Policies for Privacy Control in Social Platforms, *First Workshop on Trust and Privacy on the Social and Semantic Web*, 2009.
- [40] Geoffrey Cooper, Kieran G. Sherlock, Bob Shaw, and Luis Valente. United States Patent 7,047,288: Automated generation of an English language representation of a formal network security policy specification, 2000.
- [41] Microsoft Corporation. Privacy in Internet Explorer: [http://msdn.microsoft.com/en-ca/library/ms537343\(v=vs.85\).asp](http://msdn.microsoft.com/en-ca/library/ms537343(v=vs.85).asp). 2012.
- [42] Microsoft Corporation. Retention Tags and Retention Policies in Exchange Server: [http://technet.microsoft.com/en-us/library/dd297955\(v=exchg.150\).aspx](http://technet.microsoft.com/en-us/library/dd297955(v=exchg.150).aspx). 2013.

- [43] Open Text Corporation. LiveLink Records Management System: <http://www.opentext.com/2/global/products/products-records-management/products-opentext-records-management.htm>. 2012.
- [44] Lorrie Cranor, Marc Langheinrich, Massimo Marchiori, Martin Presler-Marshall, and Joseph Reagle. The platform for privacy preferences 1.0 (P3P1.0) specification. *W3C recommendation*, 2002.
- [45] Alcino Cunha. Bounded Model Checking of Temporal Formulas with Alloy. *arXiv preprint arXiv:1207.2746*, 2012.
- [46] Bill Curtis, Marc I. Kellner, and Jim Over. Process modeling. *Communications of the ACM*, 35(9):75–90, 1992.
- [47] Umeshwar Dayal, Barbara Blaustein, Alex Buchmann, Upen Chakravarthy, Meichun Hsu, R Ledin, Dennis McCarthy, Arnon Rosenthal, Sunil Sarin, Michael J. Carey, et al. The hipac project: Combining active databases and timing constraints. *ACM Sigmod Record*, 17(1):51–70, 1988.
- [48] S. Demri and R. Gascon. The Effects of Bounding Syntactic Resources on Presburger LTL. In *14th International Symposium on Temporal Representation and Reasoning*, pages 94–104, 2007.
- [49] Stéphane Demri. LTL over integer periodicity constraints. *Theoretical Computer Science*, 360(1):96–123, August 2006.
- [50] Stéphane Demri and Deepak D’Souza. An automata-theoretic approach to constraint LTL. *Information and Computation*, 205(3):380–415, March 2007.
- [51] Birgit Demuth and Heinrich Hussmann. Using UML/OCL constraints for relational database design. *UML 99: The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 751–751, 1999.
- [52] Birgit Demuth, Heinrich Hußmann, and Sten Loecher. OCL as a specification language for business rules in database applications. *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 104–117, 2001.

- [53] Yuetang Deng, Phyllis Frankl, and Zhongqiang Chen. Testing database transaction concurrency. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 184–195. Society Press, 2003.
- [54] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
- [55] Department of Finance Canada. Audit of Leave and Overtime - Final Audit Report, <http://www.fin.gc.ca/treas/audit/pdf/lor08-eng.pdf>.
- [56] Alin Deutsch, Richard Hull, Fabio Patrizi, and Victor Vianu. Automatic verification of data-centric business processes. In *Proceedings of the 12th International Conference on Database Theory, ICDT '09*, pages 252–267, New York, NY, USA, 2009. ACM.
- [57] Nikhil Dinesh, Aravind Joshi, Insup Lee, and Oleg Sokolsky. Checking traces for regulatory conformance. In *Runtime Verification*, pages 86–103. Springer, 2008.
- [58] Deborah D. Downs, Jerzy R. Rub, Kenneth C. Kung, and Carole S. Jordan. Issues in Discretionary Access Control. *Security and Privacy, IEEE Symposium on*, 0:208, 1985.
- [59] Eindhoven University of Technology. Declare Modeling System: <http://www.win.tue.nl/declare/>.
- [60] Ugonwa Ekweozor and Babis Theodoulidis. Review of retention management software systems. *Records Management Journal*, 14(2):65–77, 2004.
- [61] Peter Emmerson. *How to Manage Your Records: a guide to effective practice*, ISBN:0902197819. Cambridge, England: ICSA Publishing, 1989.
- [62] François Fages and Aurélien Rizk. From model-checking to temporal logic constraint solving. *Principles and Practice of Constraint Programming-CP 2009*, pages 319–334, 2009.

- [63] David Ferraiolo, Janet Cugini, and D Richard Kuhn. Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th Annual Computer Security Application Conference*, pages 241–48. sn, 1995.
- [64] David Ferraiolo and Richard Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.
- [65] L. Fischer. *BPM and Workflow Handbook: Spotlight on on Business Intelligence*, ISBN: 9780977752713. Future Strategy Publishers, 2010.
- [66] Fabian Friedrich, Jan Mendling, and Frank Puhlmann. Process model generation from natural language text. In *Proceedings of the 23rd International Conference on Advanced Information Systems Engineering, CAiSE'11*, pages 482–496, Berlin, Heidelberg, 2011. Springer-Verlag.
- [67] Denis Gagné and André Trudel. A temporal semantics for workflow control patterns. In *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC '08*, pages 999–1004, Washington, DC, USA, 2008. IEEE Computer Society.
- [68] Stella Gatzui and Klaus R. Dittrich. Samos: An active object-oriented database system. *IEEE Data Engineering Bulletin*, 15(1-4):23–26, 1992.
- [69] George M Giaglis. A taxonomy of business process modeling and information systems modeling techniques. *International Journal of Flexible Manufacturing Systems*, 13(2):209–228, 2001.
- [70] Heidi Gregersen and Christian S Jensen. Temporal Entity-Relationship models: a survey. *Knowledge and Data Engineering, IEEE Transactions on*, 11(3):464–497, 1999.
- [71] OASIS Group. *OASIS Group Specifications for WS-BPEL*: <https://www.oasis-open.org/committees/wsbpel/>.

- [72] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.
- [73] Eric N. Hanson. The design and implementation of the ariel active database rule system. *Knowledge and Data Engineering, IEEE Transactions on*, 8(1):157–172, 1996.
- [74] Eric N Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, JB Park, and Albert Vernon. Scalable trigger processing. In *15th International Conference on Data Engineering*, pages 266–275. IEEE, 1999.
- [75] Ramesh Hariharan, Madhavan Mukund, and V Vinay. Runtime monitoring of metric first-order temporal properties. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2, pages 49–60. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008.
- [76] Michael Havey. *Essential Business Process Modeling*. O’Reilly Media, 2009.
- [77] Joseph M. Hellerstein and Michael Stonebraker. Predicate Migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’93, pages 267–276, New York, NY, USA, 1993. ACM.
- [78] D.G. Hill. *Data Protection: Governance, Risk Management, and Compliance, ISBN:9781439806920*. Taylor & Francis, 2009.
- [79] Ian Hodkinson, Frank Wolter, and Michael Zakharyashev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied logic*, 106(1):85–134, 2000.
- [80] Gerard Holzmann. *Spin model checker: Primer and Reference manual, ISBN:0-321-22862-6*. Addison-Wesley Professional, 2003.
- [81] Terry Huston. Security issues for implementation of e-medical records. *Communications of the ACM*, 44(9):89–94, September 2001.

- [82] Sonia Jahid, Carl A. Gunter, Imranul Hoque, and Hamed Okhravi. MyABDAC: compiling XACML policies for attribute-based database access control. In *Proceedings of the first ACM conference on Data and application security and privacy, CODASPY '11*, pages 97–108, New York, NY, USA, 2011. ACM.
- [83] Sonia Jahid, Imranul Hoque, Hamed Okhravi, and Carl A Gunter. Enhancing database access control with XACML policy. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 130–133, 2009.
- [84] Christian S. Jensen and Richard T. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.
- [85] John Jeston and Johan Nelis. *Business Process Management: Practical guidelines to successful implementations, ISBN:9780750686563*. Routledge, 2008.
- [86] Xin Jin. Applying Model Driven Architecture approach to Model Role Based Access Control System. Master’s Thesis at the University of Ottawa, Canada, 2006.
- [87] Vaux John. SQL Server XP-CMDSHELL Reference, <http://msdn.microsoft.com/en-us/library/ms175046.aspx>, 2012.
- [88] James BD Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. A generalized temporal role-based access control model. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):4–23, 2005.
- [89] M.B. Jurič, B. Mathew, and P.G. Sarang. *Business Process Execution Language for Web Services 2nd Edition*. From Technologies to Solutions. Packt Pub., 2006.
- [90] John Karat, Clare-Marie Karat, and Carolyn Brodie. Human-Computer Interaction viewed from the intersection of Privacy, Security, and Trust. *Human-Computer Interaction: Design Issues, Solutions, and Applications*, page 311, 2009.
- [91] Nadzeya Kiyavitskaya, Nicola Zeni, Travis D. Breaux, Annie I. Antón, James R. Cordy, Luisa Mich, and John Mylopoulos. Extracting rights and obligations from regulations: Toward a tool-supported process. In *Proceedings of the 22nd IEEE/ACM*

International Conference on Automated Software Engineering, ASE '07, pages 429–432, New York, NY, USA, 2007. ACM.

- [92] Nadzeya Kiyavitskaya, Nicola Zeni, Travis D. Breaux, Annie I. Antón, James R. Cordy, Luisa Mich, and John Mylopoulos. Automating the extraction of rights and obligations for regulatory compliance. In *Proceedings of the 27th International Conference on Conceptual Modeling, ER '08*, pages 154–168, Berlin, Heidelberg, 2008. Springer-Verlag.
- [93] Ryan KL Ko, Stephen SG Lee, and Eng Wah Lee. Business process management (BPM) standards: a survey. *Business Process Management Journal*, 15(5):744–791, 2009.
- [94] Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences*, 61(2):302–332, 2000.
- [95] Angelika Kotz-Dittrich and Eric Simon. Active database systems: Expectations, commercial experience, and beyond. *Active Rules in Database Systems*, pages 367–404, 1999.
- [96] Jacob Lamm, Sumner Blount, Steve Boston, Marc Camm, Robert Cirabisi, Nancy Cooper, Galina Datskovsky, Christopher Fox, Kenneth Handal, William McCracken, John Meyer, Helge Scheil, Alan Srulowitz, and Rob Zanella. *Under Control: Governance Across the Enterprise, ISBN:1430215933*. Apress, Berkely, CA, USA, 1st edition, 2009.
- [97] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.
- [98] Daniel F. Lieuwen, Narain H. Gehani, and Robert M. Arlein. The Ode Active Database: Trigger Semantics and Implementation. In *Proceedings of the Twelfth International Conference on Data Engineering, ICDE '96*, pages 412–420, Washington, DC, USA, 1996. IEEE Computer Society.

- [99] Beate List and Birgit Korherr. An evaluation of conceptual business process modelling languages. In *Proceedings of the 2006 ACM Symposium on Applied Computing, SAC '06*, pages 1532–1539, New York, NY, USA, 2006. ACM.
- [100] Ruopeng Lu and Shazia Sadiq. A survey of comparative business process modeling approaches. In *Proceedings of the 10th International Conference on Business Information Systems, BIS'07*, pages 82–94, Berlin, Heidelberg, 2007. Springer-Verlag.
- [101] Fabrizio Maria Maggi, Michael Westergaard, Marco Montali, and Wil M. P. van der Aalst. Runtime verification of LTL-Based declarative process models. In *Proceedings of the Second International Conference on Runtime verification, RV'11*, pages 131–146, Berlin, Heidelberg, 2012. Springer-Verlag.
- [102] Jan Mendling, Hajo A. Reijers, and Wil van der Aalst. Seven process modeling guidelines (7pmg). *Information and Software Technology*, 52(2):127–136, 2010.
- [103] James Bret Michael, Vanessa L. Ong, and Neil C. Rowe. Natural-language processing support for developing policy-governed software systems. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, TOOLS '01, Washington, DC, USA, 2001. IEEE Computer Society.
- [104] James Bret Michael, Edgar H. Sibley, Richard F. Baum, and Fu Li. On the axiomatization of security policy: some tentative observations about logic representation. In *Results of the Sixth Working Conference of IFIP Working Group 11.3 on Database Security on Database security, VI : status and prospects: status and prospects*, pages 367–386, New York, NY, USA, 1993. Elsevier Science Inc.
- [105] Bernard Moulin and Daniel Rousseau. Automated knowledge acquisition from regulatory texts. *IEEE Expert: Intelligent Systems and Their Applications*, 7(5):27–32, October 1992.
- [106] Bernard Moulin and Daniel Rousseau. SACD: a system for acquiring knowledge from regulatory texts. *Computers and Electrical Engineering*, 20(2):131–149, 1994.

- [107] Ranga Narasimhan. Auto generate SQL Server UPDATE triggers for data auditing: <http://www.mssqltips.com/sqlservertip/1770/auto-generate-sql-server-update-triggers-for-data-auditing/>. 2009.
- [108] T. Neward. Interoperability Happens: The Vietnam of Computer Science. *Ted Neward's Technical Blog*: <http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx>, 2006.
- [109] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, July 2003.
- [110] Sejong Oh and Seog Park. Task-role-based access control model. *Information Systems*, 28(6):533–562, 2003.
- [111] Lars E. Olson, Carl A. Gunter, and P. Madhusudan. A formal framework for reflective database access control policies. In *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS '08*, pages 289–298, New York, NY, USA, 2008. ACM.
- [112] OMG. *BPMN v2.0 by Example*, OMG: www.omg.org/spec/BPMN/20100601/10-06-02.pdf.
- [113] Object Management Group (OMG). *Object Management Group (OMG): Specifications available at: <http://www.omg.org/spec/BPMN/2.0/>*.
- [114] Object Modeling Group (OMG). *UML/OCL v2.0: Specifications available at: <http://www.omg.org/technology/documents/formal/ocl.htm>*.
- [115] Elizabeth J. O’Neil. Object/relational mapping 2008: Hibernate and the entity data model (EDM). In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08*, pages 1351–1356, New York, NY, USA, 2008. ACM.
- [116] Vanessa L. Ong. *Masters Thesis: An Architecture and Prototype System for Automatically Processing Natural-Language Statements of Policy*. Storming Media, 2001.

- [117] Stewardship Ontario. ECO Fees and Charges at Retail Locations, URL: <http://www.stewardshipontario.ca/consumers/what-we-do/mhswenvironmentalfees>.
- [118] Joël Ouaknine and James Worrell. Some recent results in metric temporal logic. *Formal Modeling and Analysis of Timed Systems*, pages 1–13, 2008.
- [119] Norman W Paton and Oscar Díaz. Active database systems. *ACM Computing Surveys (CSUR)*, 31(1):63–103, 1999.
- [120] Maja Pesic, H. Schonenberg, Natalia Sidorova, and Wil van der Aalst. Constraint-based workflow models: Change made easy. In *Proceedings of the 2007 OTM Confederated International Conference on The move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I*, OTM’07, pages 77–94, Berlin, Heidelberg, 2007. Springer-Verlag.
- [121] Maja Pesic and Wil van der Aalst. A declarative approach for flexible business processes management. In *Business Process Management Workshops*, pages 169–180. Springer, 2006.
- [122] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.
- [123] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.
- [124] Matteo Pradella. A User’s Guide to Zot. *arXiv preprint arXiv:0912.5014*, 2009.
- [125] Matteo Pradella, Angelo Morzenti, and Pierluigi San Pietro. A metric encoding for bounded model checking (extended version). *CoRR*, abs/0907.3085, 2009.
- [126] Matteo Pradella, Angelo Morzenti, and Pierluigi San Pietro. Bounded satisfiability checking of metric temporal logic specifications. *ACM Transactions in Software Engineering Methodologies*, 2012.
- [127] Public Works and Government Services Canada. Frequently Asked Questions: Pay Modernization Project - <http://www.tpsgc-pwgsc.gc.ca/remuneration-compensation/txt/tap-tpa-faq-eng.html>.

- [128] Nicolas Racz, Edgar Weippl, and Andreas Seufert. A frame of reference for research of integrated Governance, Risk and Compliance (GRC). In *Proceedings of the 11th IFIP TC 6/TC 11 International Conference on Communications and Multimedia Security*, CMS'10, pages 106–117, Berlin, Heidelberg, 2010. Springer-Verlag.
- [129] Zahid Rashid, Abdul Basit, and Zahid Anwar. TRDBAC: Temporal Reflective Database Access Control. In *6th International Conference on Emerging Technologies (ICET)*, pages 337–342, 2010.
- [130] Joseph Reagle and Lorrie Faith Cranor. The Patform for Privacy Preferences. *Communications of the ACM*, 42(2):48–55, 1999.
- [131] Robert W. Reeder, Clare-Marie Karat, John Karat, and Carolyn Brodie. Usability challenges in security and privacy policy-authoring interfaces. In *Proceedings of the 11th IFIP TC 13 International Conference on Human-Computer Interaction - Volume Part II*, INTERACT'07, pages 141–155, Berlin, Heidelberg, 2007. Springer-Verlag.
- [132] Riva Ricmond. NY Times: A Loophole Big Enough for a Cookie to Fit Through. 2010.
- [133] Kristin Y. Rozier and Moshe Y. Vardi. LTL Satisfiability Checking. In *Proceedings of the 14th International SPIN Conference on Model Checking Software*, pages 149–167, Berlin, Heidelberg, 2007. Springer-Verlag.
- [134] J. Russell and R. Cohn. *Enterprise Privacy Authorization Language*, ISBN: 9785512195864. 2012.
- [135] Ksenia Ryndina, Jochen M. Küster, and Harald Gall. Consistency of business process models and object life cycles. In *Proceedings of the 2006 International Conference on Models in Software Engineering*, MoDELS'06, pages 80–90, Berlin, Heidelberg, 2006. Springer-Verlag.
- [136] Ravi Sandhu and Jaehong Park. Usage control: A vision for next generation access control. *Computer Network Security*, pages 17–31, 2003.

- [137] Ravi S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.
- [138] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.
- [139] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.
- [140] Peter Pin shan Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.
- [141] Leilei Shi and David W. Chadwick. A controlled natural language interface for authoring access control policies. In *Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11*, pages 1524–1530, New York, NY, USA, 2011. ACM.
- [142] John W. Sias, Wen-Mei W. Hwu, and David I. August. Accurate and efficient predicate analysis with binary decision diagrams. In *Proceedings. 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 112–123, 2000.
- [143] M.G. Silverman. *Compliance Management for Public, Private, or Non-Profit Organizations, ISBN:9780071496407*. McGraw-Hill Education, 2008.
- [144] Eric Simon and Angelika Kotz Dittrich. Promises and realities of active database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases, VLDB '95*, pages 642–653, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.
- [145] Avik Sinha, Amit Paradkar, Palani Kumanan, and Branimir Boguraev. An Analysis Engine for Dependable Elicitation on Natural Language Use Case Description and its Application to Industrial Use Cases. In *IBM Technical Report*, 2008.
- [146] Avik Sinha and Paradkar Paradkar. Use Cases to Process Specifications in Business Process Modeling Notation. In *2010 IEEE International Conference on Web Services (ICWS)*, pages 473–480, july 2010.

- [147] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM (JACM)*, 32(3):733–749, 1985.
- [148] Morris Sloman and Emil Lupu. Security and management policy specification. *Network, IEEE*, 16(2):10–19, 2002.
- [149] Frank Stajano. A Gentle Introduction to Relational and Object Oriented Databases. *Olivetti & Oracle Research Laboratory, Cambridge, UK*, 1998.
- [150] Jones Steve. Return Values from XP-CMDSHELL, 2010.
- [151] Anthony Tarantino. *The Governance, Risk, and Compliance Handbook: Technology, Finance, Environmental, and International Guidance and Best Practices*, ISBN:04700958. Wiley Press, Hoboken, NJ, USA, 2008.
- [152] SQL Anywhere Engineering Team. Query Processing Based on SQLAnywhere 12.0.1 Architecture, Mar 2012.
- [153] Bernhard Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology*, ISBN:3540654704. Springer, 2000.
- [154] David Toman and Jan Chomicki. Implementing temporal integrity constraints using an active DBMS. In *Active Database Systems: Proceedings of the Fourth International Workshop on Research Issues in Data Engineering*, pages 87–95. IEEE, 1994.
- [155] Treasury Board of Canada Secretariat Canada. Treasury Board Policy Suite - <http://www.tbs-sct.gc.ca/pol/index-eng.aspx>.
- [156] CyLab: Carnegie Mellon University. 2006 Privacy Policy Trends Report. 2006.
- [157] Process Mining Group Eindhoven Technical University. ProM: Process Modeling Workbench.
- [158] Wil van der Aalst. Making Work Flow: On the Application of Petri Nets to Business Process Management. In *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets, ICATPN '02*, pages 1–22, London, UK, UK, 2002. Springer-Verlag.

- [159] Wil van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
- [160] Wil Van der Aalst, Marlon Dumas, A Ter Hofstede, Nick Russell, H Verbeek, and Petia Wohed. Life after BPEL? *Formal Techniques for Computer Systems and Business Processes*, pages 35–50, 2005.
- [161] Wil van der Aalst, Maja Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - Research and Development*, 23:99–113, 2009.
- [162] Wil van der Aalst, Arthur ter Hofstede, and Mathias Weske. Business process management: A survey, isbn:3540403183. *Business Process Management*, pages 1019–1019, 2003.
- [163] Wil van der Aalst, Arthur H. M. Ter Hofstede, Bartek Kiepuszewski, and Alistair P Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.
- [164] Wil van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, Cambridge, MA, USA, 2004.
- [165] Kami Vaniea, Clare-Marie Karat, Joshua B. Gross, John Karat, and Carolyn Brodie. Evaluating assistance of natural language policy authoring. In *Proceedings of the 4th Symposium on Usable Privacy and Security, SOUPS '08*, pages 65–73, New York, NY, USA, 2008. ACM.
- [166] Moshe Y. Vardi. Constraint satisfaction and Database Theory: a tutorial. In *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 76–85. ACM, 2000.
- [167] Moshe Y. Vardi. Model checking for database theoreticians. In *Database Theory, ICDT 2005*, pages 1–16. Springer, 2005.
- [168] Veterans Affairs Canada. Employee Compensation Audit, <http://www.veterans.gc.ca/pdf/deptReports/2010-may-eca.pdf>.

- [169] Toby Walsh. SAT v CSP. *Principles and Practice of Constraint Programming (CP 2000)*, pages 441–456, 2000.
- [170] Stephen A White and Derek Miers. *BPMN modeling and reference guide*, ISBN: 0977752720. Future Strategies Inc., 2008.
- [171] Jennifer Widom. The starburst active database rule system. *Knowledge and Data Engineering, IEEE Transactions on*, 8(4):583–595, 1996.
- [172] Petia Wohed, Wil van der Aalst, Marlon Dumas, Arthur ter Hofstede, and Nick Russell. On the suitability of BPMN for Business Process Modelling. *Business Process Management*, pages 161–176, 2006.
- [173] Frank Wolter and Michael Zakharyashev. Spatio-temporal representation and reasoning based on rcc-8. In *In Proceedings of the 7th Conference on Principles of Knowledge Representation and Reasoning, KR2000*, pages 3–14. Morgan Kaufmann, 2000.
- [174] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12*, pages 12:1–12:11, New York, NY, USA, 2012. ACM.
- [175] Xinwen Zhang, Masayuki Nakae, Michael J. Covington, and Ravi Sandhu. A usage-based authorization framework for collaborative computing systems. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, pages 180–189. ACM, 2006.
- [176] Jingren Zhou, P.-A. Larson, J. Goldstein, and Luping Ding. Dynamic Materialized Views. In *23rd IEEE International Conference on Data Engineering*, pages 526–535, 2007.
- [177] Daniel C Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M Lohman, Roberta J Cochrane, Hamid Pirahesh, Latha Colby, Jarek Gryz, Eric Alton, et al.

- Recommending materialized views and indexes with the IBM DB2 design advisor. In *International Conference on Autonomic Computing*, pages 180–187. IEEE, 2004.
- [178] Geraldo Zimbrão, Rodrigo Miranda, Jano Moreira de Souza, Mario H Estolano, and Francisco P Neto. Enforcement of business rules in relational databases using constraints. In *Proceedings of XVIII Simposio Brasileiro de Bancos de Dados/SBBD 2003*, pages 129–141, 2003.
- [179] Michael zur Muehlen and Danny T Ho. Service process innovation: a case study of BPMN in practice. In *Proceedings of the 41st annual International Conference on System Sciences*, pages 372–372. IEEE, 2008.
- [180] Michael zur Muehlen and Jan Recker. How much language is enough? Theoretical and practical use of the business process modeling notation. In *Advanced Information Systems Engineering*, pages 465–479. Springer, 2008.