Developing and Executing Electronic Commerce Applications with Occurrences

Alan Samuel Abrahams

Peterhouse
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

September 2002

Abstract

To provide a more direct mapping of business process specifications to software implementations, the next generation of enterprise workflow systems must move away from a procedural execution style and towards an *event-driven model* that monitors and controls the business process in accordance with a periodically changing set of stored *business contracts, intra-organizational policies, and legislative requirements*.

It is the thesis of this dissertation that a crucial, and hitherto neglected, aspect of electronic commerce application development techniques and tools is the *analysis, modelling, storage, and interrogation* of the *business occurrences and contractual provisions* that drive workflow applications. Extant systems for event monitoring, business rules, policy-based management, contracting, and workflow execution do not directly represent, store, enact and enforce the subtle and often-times conflicting contractual and regulatory provisions contained in business requirements specifications. Explicit treatments of fundamental legal conceptions such as obligations, permissions, and powers are absent from conventional software. Furthermore, previous work in the event-, rule-, and policy-based execution styles overlooks the early phases of system development: current approaches lack guidelines to allow analysts to transform English-language specifications of contracts, policies, laws, and regulations into a structured form suitable for direct input into an implementation environment, making seamless transition through the system development life cycle a far distant dream.

In order to address these issues, this dissertation presents a novel *occurrence-based development approach and execution infrastructure*. The main contribution of this thesis is *a method and infrastructure to create executable and queryable specifications for electronic commerce applications*. A persistent history of business occurrences and associated contractual implications is kept. Multi-phase assistance for the systems development life cycle of business workflow applications is provided: we propose an analysis method coupled with implementation support in the form of a software environment, information model, algorithms, and interfaces. Our method embeds fundamental legal conceptions and integrates diverse theories from the philosophical literature on jurisprudence, deontic logic, knowledge representation, speech acts, and event semantics. A powerful and generic database wrapper service delivers the functionality required by businesses to store, query, execute, monitor, enforce, and reason about the prescriptive and descriptive norms that govern the behaviour of their software and human activity systems. As a complement to the run-time infrastructure, specification-time facilities to detect and resolve inconsistencies between subjective and possibly conflicting clauses are incorporated.

The theoretical concepts illustrated in this dissertation are implemented in a Java-based prototype, EDEE, which provides active database functionality atop arbitrary passive relational data stores. This thesis describes the architecture of the occurrence-based Electronic-commerce Development and Execution Environment (EDEE), and presents persuasive examples demonstrating structured analysis and implementation of realistic English-language requirements specifications using our new mechanisms.

To my parents
who I love beyond words.
I hope you'll accept these 58,904 as a start.

And to my dear departed gogo
May Kubheka
1930 - 2002
who, though technically lacking the legal authority, would daily page me across the house at 35 Grosvenor Crescent, Durban North, with yells of "Dr Uh-luh ! ... Dr Uh-luh !!".
A personality lesser known than Mandela and Tutu, but no less influential. Her words remain always a forceful symbol of the wisdom and spirit of a traditional South African Mother: what is lacked in legal power, is more than recompensed in motivational charge for the youth.
Those pages were for me; these are for you.

Hamba Kahle, Mama...

Preface

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text. The pronouns 'we' and 'our' in the text, which have been used for stylistic reasons, should be taken to refer to the singular author.

This dissertation is not substantially the same as any that I have submitted for a degree or diploma or any other qualification at any other university.

No part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed 60,000 words including tables and footnotes, but excluding appendices, bibliography, and diagrams.

Conventions Used

- Code excerpts are shown in `Courier font`.
- Sentences of predicate logic are shown in *italic Times Roman*.
- Data items are shown in Arial 8 point font.
- Unstructured textual user specifications, including excerpts from business contracts, policies, and regulations, are shown in Arial 9 point font.
- At their first mention in the text, product and company names are shown in **bold**.
- Internet addresses (Uniform Resource Locators) are shown in Arial Narrow font.
- Requirements for the research deliverable, which are identified in the Analysis of Related Work (Chapter 2), are indicated boxed and in bold, and are numbered for cross-reference:

Requirement 16	**Asynchronous fulfilment of chosen obligations must be supported as an enforcement style**

In the chapters that describe the contributions of this work, greyed, right-aligned sidebars map the particular solutions (performances) back to specific requirements (obligations). This provides *traceability* from identified requirements to implemented solutions. For example:

…

In the EDEE architecture, components consult the database and choose which obligations to fulfil.

…

☑ **Requirement 16 (pg 34):** Asynchronous fulfilment of chosen obligations must be supported as an enforcement style

Acknowledgements

Heartfelt thanks are due to a number of people who made this possible.

To my family in South Africa, London, and Manchester for always thinking of me.

I would like to thank my supervisor, Jean Bacon and colleagues in the Opera group and Computer Laboratory, for their support, encouragement, and critical insights during the completion of this work. I enjoyed the warm hospitality of Professor Steve Kimbrough, and the faculty and administrative staff of the Department of Operations and Information Management at the University of Pennsylvania's Wharton Business School, who graciously hosted me for two months in Spring 2002. I also owe a great debt to the management and staff of Dimension Data plc's iCommerce Internet Services division, who allowed me to spend three months at their Johannesburg head offices analysing user requirements specifications during the Summer of 2000.

A number of people very kindly provided me with hard copies of publications, feedback, pointers, and/or simply warm encouragement and welcoming. Thank you to Krishnan Anand, José Carmo, Aspassia Daskalopulu, Theo Dimitrakos, John Dobson, Huw Evans, Ben Grosof, Andrew Jones, Ron Lee, Tom Lee, David Makinson, Miriam Masullo, Tom Maibaum, Roderick Munday, Filipé Santos, Marek Sergot, Lewis Tiffany, Marge Weiler, DJ Wu, and Carina de Villiers. Mark Spiteri helpfully contributed the *Microsoft Word* template used to format this document. Also, I appreciate the invaluable technical support assistance provided by our local `sys-admins`, especially Ian Grant. Nathan Dimmock, Martyn Johnson, Ian Pratt, and the friendly folk in the Systems Research Group kindly volunteered hardware for the experiments.

I will be forever grateful to my friends and colleagues in Cambridge and Philadelphia for their warmth and friendship during these three years away from home. To my dear friend Biagio 'Mamzer' Mazzi, who promised to mention me in his thesis if I mentioned him in mine; the utterance of the first half of this sentence renders the conditional obligation unconditional. To Nathan Dimmock, David Eyers, Aras 'Kimbo' Kimis, Naila Mimouni, Brian Shand, and my dad, who took the time to read and comment on the drafts. Dave is credited with the beautiful typesetting of the two diagrams of query parse trees. To the Opera tea crowd, and to Cala, Galya, Gerrit, Kerry, Liv, Marc, Mon, Nick, Orit, Pete, Ravi, Steph, Ted, and Wojciech for their wonderful company during these three years.

Finally, I am grateful to the Cambridge Commonwealth Trust, Overseas Research Students Scheme, and University of Cape Town for funding this research. I appreciate also the financial support provided by the Cambridge Philosophical Society, the CaberNet Network of Excellence in Distributed and Dependable Computing Systems, the University of Glasgow, King's College London, the University of Cambridge Computer Laboratory, Microsoft Research Cambridge, and Peterhouse, which allowed me to travel to various conferences, workshops, research visits, and invited talks during the course of my studies.

Publications

Aspects of the work described in this dissertation feature in the following papers:

[AB2000] Abrahams AS and Bacon JM. Event-centric Business Rules in E-commerce Applications. *Workshop on Best Practices in Business Rule Design and Implementation at the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*. Minneapolis, MN. October 2000.

[AB2001a] Abrahams AS and Bacon JM. Event-centric Policy Specification for E-commerce Applications. *Policy 2001, Workshop on Policies for Distributed Systems and Networks*. Bristol, UK. January 2001.

[AB2001b] Abrahams AS and Bacon JM. Occurrence-centric Policy Specification for E-commerce Applications. *Workshop on Formal Modelling for Electronic Commerce (FMEC 2001)*. Norway, Oslo. June 2001.

[AB2001c] Abrahams AS and Bacon JM. Representing and Enforcing Electronic Commerce Contracts over a Wide Range of Platforms Using Occurrence Stores. *4th CaberNet Plenary Workshop*. Pisa, Italy. October 2001.

[AB2001d] Abrahams AS and Bacon JM. Representing and Enforcing E-Commerce Contracts Using Occurrences. *Proceedings of the 4th International Conference on Electronic Commerce Research (ICECR4)*. Edwin L. Cox School of Business, Southern Methodist University. Dallas, TX. November 2001.

[AB2002a] Abrahams AS and Bacon JM. A Software Implementation of Kimbrough's Disquotation Theory for Representing and Enforcing Electronic Commerce Contracts. *Group Decision and Negotiations Journal*. 11(6). Special Issue on Formal Modelling in Electronic Commerce. INFORMS. pp. 1-38. November 2002.

[AB2002b] Abrahams AS and Bacon JM. The Life and Times of Identified, Situated, and Conflicting Norms. *Sixth International Workshop on Deontic Logic in Computer Science (DEON'02)*. Imperial College, London, UK. May 2002.

[AEB2002a] Abrahams AS, Eyers DM, and Bacon JM. A Coverage Determination Mechanism for Checking Business Contracts against Organizational Policies. *3rd VLDB Workshop on Technologies for E-Services (TES'02)*. Hong Kong, China. August 2002. Lecture Notes in Computer Science 2444. Springer-Verlag. Berlin, Germany. pp. 97-106. 2002.

[AEB2002b] Abrahams AS, Eyers DM, and Bacon JM. Mechanical Consistency Analysis for Business Contracts and Policies. *Proceedings of the 5th International Conference on Electronic Commerce Research (ICECR5)*. Montreal, Canada. October 2002.

Publications

[AEB2002c] Abrahams AS, Eyers DM, and Bacon JM. An asynchronous rule-based approach for business process automation using obligations. *Proceedings of the 3rd ACM SIGPLAN Workshop on Rule-Based Programming (RULE'02)*. Pittsburgh, PA. October 2002.

[AK2002] Abrahams AS and Kimbrough SO. Treating Disjunctive Obligation and Conjunctive Action in Event Semantics with Disquotation. *Wharton Business School Working Paper Series*. University of Pennsylvania. Philadelphia, PA. 2002.

[DA99] De Villiers C and Abrahams AS. A Model for Addressing the Development of Electronic Commerce Applications in Information Systems Courses. *International Academy for Information Management, 14th Annual Conference (IAIM 1999)*. Charlotte, NC. December 10-12, 1999.† Also published under the title 'A Model for Teaching the Development of Electronic Commerce Applications' in *Journal of Informatics Education & Research*. 2(1). pp. 1-8. Spring 2000.

This work was also presented at invited talks at:

- The School of Informatics, *University of Pretoria*, South Africa
- The Department of Computer Science, *University of Glasgow*, Scotland
- The Department of Computer Science, *King's College London*, England
- The Department of Operations and Information Management, Wharton Business School, *University of Pennsylvania*, USA; and
- The Department of Information Systems, Faculty of Commerce, *University of Cape Town*, South Africa

† Awarded Best Paper Overall of conference.

Contents

Index of Requirements

The following requirements are identified in Chapter 2 (Analysis of Related Work) and addressed in the remaining chapters:

Requirement			Addressed
No	(Pg)	Description	Pg
1	(19)	The information model must be able to describe occurrences of past events, states, and processes, as well as the (active and passive) role-players involved.	79
2	(20)	Occurrences with nested propositional content must be expressible.	127
3	(22)	Matching (occurrence detection) must be against a long history.	90
4	(22)	Persistent storage of business-level occurrences is required.	79
5	(23)	Business applications require occurrences to be accessible via ad-hoc queries, rather than via fixed access paths.	82
6	(24)	A business-level, rather than technical-system-level, approach is needed for business process automation.	80
7	(28)	Situations should be interpreted against a dynamic set of rules.	91
8	(29)	A large and growing number of machine-enforceable rules should be controlled and managed in a database, not haphazardly distributed in text files.	120
9	(29)	Analytic conflict detection is desirable.	174
10	(30)	Pattern-pattern matching functionality is required for analytic policy conflict detection.	90
11	(32)	Fundamental legal conceptions – such as duties, privileges, powers, and immunities [Hoh78] – must be natively incorporated in the development approach.	166
12	(32)	Rule attributes – such as author, specification time and document or utterance location, scope, and jurisdiction – should be stored, as should attributes of entities related to rules – such as author's roles over time.	170
13	(32)	Reasoning techniques should emulate legal reasoning. Conflict resolution facilities should allow selection of applicable rules based on recency, specificity, location, authority, or other criteria [GLC99].	189
14	(32)	Context-specific rule precedence must be supported.	181
15	(33)	The treatment of obligations – whether to void or violate them in a given case – must be user-definable.	190
16	(34)	Asynchronous fulfilment of chosen obligations must be supported as an enforcement style	200
17	(38)	It must be possible to define and store the criteria that an item must satisfy – the description (query) it must fit – in order for it to count as being of a certain type.	89
18	(39)	We must be able to express policies applying to intensionally described (rather than merely extensionally listed) groups of objects. The members of such groups may change dynamically.	89
19	(39)	Policy environments require a coverage detection service and interface that would allow objects to determine which of a changing set of descriptions they, or other objects, fall under.	205

Index of Requirements

Requirement No	(Pg)	Description	Addressed Pg
20	(41)	Though it is perhaps not fully machine-automatable, the progression from English specifications to machine interpretable policies should be further systematized.	117
21	(42)	Impersonal (collective) obligations must be expressible and checkable.	160
22	(43)	It must be possible to express and monitor both obliged *actions* and obliged *states of affairs*.	158
23	(43)	The implementation must allow introduction of broad reaching provisions (e.g. defining 'fulfilment' and 'violation') by a single insertion anywhere in the specification.	163
24	(44)	We must be able to refer to individual obligations of each party, and trace each obligation to the general prescriptions, and events, that brought it about, or that terminated it.	156
25	(45)	Physical occurrences must be distinguished from legal occurrences. Both must be recorded.	133
26	(49)	Provisions (legal relations), should be explicitly stored, not implicitly encoded in process models.	166
27	(50)	Existence of obligations must be derived from interpretation of law, rather than from a closed set of communicative acts or a rigidly defined protocol.	159
28	(51)	The model should provide fundamental legal conceptions, rather than hard-code the constraints of a particular system of law.	167
29	(60)	Provisions, whether emanating from contracts, policies, or laws (inter-, intra-, or extra-organizational provisions), should be uniformly represented, to facilitate consistency checking.	121
30	(62)	An approach is needed where provisions are explicitly captured as data, and are thus readily available for inspection and analysis.	166
31	(67)	Directed obligations, with actor, beneficiary, liable party, source utterance, and issuer, must be expressible.	154
32	(67)	Obligations should be individually identified.	158
33	(70)	Obligation life-cycle must be modelled. Obligations must be traceable to the events, states, and regulations that brought them about or cancelled them out.	161
34	(70)	Both once-off and persistent rights should be expressible.	141
35	(70)	Primitives governing change of legal relations – such as power and immunity [Hoh78] – must be provided.	137

List of Abbreviations

ACE	Attempto Controlled English [FSS98, FSS99]
ACL	Agent Communication Language
ADEPT	Advanced Decision Environment for Process Tasks [JFN2000]
ASL	Authorisation Specification Language [JSS97]
BCA	Business Contract Architecture [MBBR95, Mil95]
CEA	Cambridge Event Architecture [BMBH2000]
ECA	Event-Condition-Action [PD99]
EDEE	E-commerce Development and Execution Environment [AB2002a, AB2001d]
FLBC	Formal Language for Business Communication [KM97, Moo2000]
GEM	Generalized Event Monitor [MS97]
IETF	Internet Engineering Task Force (http://www.ietf.org/)
LaSCO	Language for Security Constraints on Objects [HPL99]
LDAP	Lightweight Directory Access Protocol (see [UM96])
LGI	Law Governed Interaction [MU2000]
NLP	Natural Language Processing [All95]
OASIS	(1) Open Architecture for Secure Interworking Services [BMBH2001] (2) Organization for the Advancement of Structured Information Standards (http://www.oasis-open.org/)
OCL	Object Constraint Language [OMG2001]
ODL	Object Definition Language (http://www.odmg.org/)
OMG	Object Management Group (http://www.omg.org/)
OPSS	Orchestra Process Support System [CDNF2001]
OQL	Object Query Language (http://www.odmg.org/)
PCIM	IETF's Policy Core Information Model [MESW2001]
PDL	(1) Bell's Policy Description Language [CL2001, CLN2000, LBN99] (2) Koch's Policy Definition Language [Koc97]
PSL	Process Specification Language [SGTV2000]
RM-ODP	Reference Model on Open Distributed Processing [ISO95]
SCLPs	Situated Courteous Logic Programs [GLC99, IBM2001, RGW2002]
SDL	Standard Deontic Logic (see [vW51, Che80, MW93])
SLA	Service Level Agreement
SLAPD	Standalone LDAP Daemon [UM96]
SPL	Security Policy Language [RZF2001]
SQL	Structured Query Language [ISO99a]
TriGS	Trigger system for GemStone [KRR98]
tpaML	Trading Partner Agreement Markup Language [DDKL2001]
UML	OMG Unified Modelling Language [OMG2001]
WfMC	Workflow Management Coalition (http://www.wfmc.org/)
WFMS	Workflow Management System
XML	Extensible Markup Language (http://www.w3c.org/xml/)

Index of Figures

Index of Figures

Index of Tables

Index of Tables

Chapter 1

Introduction

A constant stream of contracts, policies, and regulations describe and prescribe the behaviour of human and automated systems in an organization. The provisions of these specifications continually mix and coalesce in subtle and complex ways. Modern e-commerce application development techniques lack the facilities to capture and interpret these dynamically changing specifications. Instead, they hard-wire them in compiled invocation sequences or rule sets. Business processes are calcified, and software hardens. Meanwhile, the crucial legal conceptions conveyed in the original expressions of requirements are diluted, and are lost as they dissolve into an opaque, stagnant pool of code.

Current e-commerce application development approaches do not provide structured representations of the continual flow of provisions emanating from contracting with partner organizations, internal policy-making, and promulgation of regulations by external authorities. They do not attend to the upstream phases of the system development life cycle: the rules-of-thumb employable by a business analyst to analyse English-language requirements specifications, in terms consistent with the implementation technology, are not made explicit. Downstream, they do not provide software facilities tailored to storing, interrogating, monitoring, and enforcing those ever-changing requirements. An ability to assess the legal consequences of operational fulfilment and environmental occurrences in the business, based on the recorded provisions, is absent. Furthermore, the lack of a persistent, historical store

of salient commercial occurrences has constrained our ability to build workflow environments that monitor and direct system execution through the *interpretation of business contracts, policies, and regulations*.

It is the thesis of this dissertation that a fresh approach, grounded in linguistics, philosophy, and law can provide the facilities required for *contract-driven execution* that are lacking in modern e-commerce application development environments. We argue that stored abstractions of normative and operational occurrences present a novel device for developing and executing e-commerce applications. Our suggestions are aimed at providing a seamless application development process, and more fluid workflow applications, which are responsive to the periodically changing directives that govern the organization's human and automated systems.

This first chapter introduces the notion of business contracts, policies, and regulations and the prospect of using them as a mechanism for executable requirements specification. Section 1.1 introduces the background to this research. An example application scenario that is used for illustration throughout this thesis is provided (Section 1.2). Section 1.3 highlights the research issues and motivation. We then list the goals and tasks that the research set out to achieve (Section 1.4) and describe the contributions of the research (Section 1.5). We conclude this introductory chapter with an outline of the structure of the rest of this document.

1.1 Research Background

In earlier work on e-commerce application development undertaken with De Villiers [DA99], we prescribed a requirements elicitation methodology based on Checkland's Soft Systems Approach [CT96a, CT96b, FJ91]. There we showed that, even in the very first stages of the system development life cycle, attention is paid to the *roles* of various stakeholders, their *norms* of behaviour, and the distinction between *actual* and *ideal* behaviour. However, our approach in that work was semi-structured, informal, and independent of development technologies.

Investigation subsequently began into the development of guidelines and focused technologies to support the progression from the specification and analysis of norms in business contracts, to the implementation of contract-driven executable systems. The results of this investigation are documented in this thesis.

The Cambridge English Dictionary [CIDE95] defines a norm as 'an accepted way of behaving'. More specifically, we may say that:

> A **norm** (given in a textual or verbal **provision**) specifies
> a set of occurrences that are acceptable or unacceptable.

Acceptable occurrences are defined in permissions, powers, and obligations. Permissions specify which occurrences do not bring about violations. Powers describe occurrences that are accepted as bringing about legal relations. Obligations refer to occurrences that are not only accepted but, more strongly, are *required*. Absence of the required occurrences is unacceptable. Unacceptable occurrences are also defined in prohibitions.

Requirements may be based on ethics, law, culture, business policies, organizational commitments, and other sources of norms. Requirements either prescribe certain actions or constrain the set of possible actions [KRR97]. At present, business policies are typically recorded in multiple hard-copy contracts with customers, suppliers, employees and partner organisations. Internal contracts are captured in procedure manuals and user requirement specifications. There is a need for formalisation of policies in order to provide an unambiguous interpretation of them [SD2000]. In e-business, the ideal is that these policies are captured precisely by the implementation code-base of the company. The progress of business should be governed by monitoring what takes place against a precise representation of a periodically changing set of contractual provisions.

The semantic form of business contracts is lost during the translation to conventional object-oriented and procedural software. Here, methods are invoked in a hard-coded sequence. The introduction of new contractual provisions requires a tedious process of search and inject, in which appropriate method-invocations must

be manually inserted into all relevant invocation sequences. Just as normalized databases ensure data consistency across applications and remove dangerous redundancy, a database of contractual provisions gives the same policy consistency benefits. Redundant dispersion of provisions across multiple SQL triggers on various tables or across multiple Java methods in various classes can be averted by a database store.

Languages like Java and C++ suffer from embedding policy in both procedural code and scattered SQL statements, making policy difficult to extract, reconcile, and modify. Business policies are often blurred by code, making it difficult to cope with their dynamic nature; they cannot easily be factored out or automatically shared by and imposed on all applications [KRR97].

In traditional approaches, specifications and code are decoupled. Policies are often implicit or haphazardly distributed in opaque code. The system's specification is not readily recollectable or modifiable. The mismatch between analysis methodologies and implementation technologies means that documentation is frequently out of synchronization with implementation. The code itself is often the only up-to-date documentation of a program [FF94], and is in a form inaccessible to management. Translating design models written in complex modelling notations such as UML [OMG2001], to implementation languages (like Java and C++) and database languages (like OQL and SQL) is time-consuming and error-prone. UML and object-oriented implementation languages do not provide constructs to model the rights and duties of parties, which are the foundations of commercial exchanges.

The semantics of obligations, powers and authority are lost in sequential invocation paradigms. It is therefore not possible to automatically assess the mutual obligations of participants who are subject to contract. Conventional Interface Definition Languages (IDLs) define available method names, and their argument and return types, but do not provide a way to specify pre- and post-conditions for method invocation. This absence of semantics makes it impossible to relate program state to the business process. These limitations point to the need to adopt a *contract-aware*, *event-driven* paradigm to monitor and control workflow execution.

1.2 Application Scenario

To clarify the problems we will address, consider the following application scenario, which we return to throughout this thesis to illustrate the plausibility of our implementation.

SkyHi Builders is a construction company. Steelmans Warehouse is a supplier of high-grade steel. SkyHi, having recently won a tender to build a new office block, enters into a contract with Steelmans. An excerpt appears as follows:

Contract between SkyHi and Steelmans entered into on 1st August 2001

…

"steel" shall mean low-carbon steel of the type Fe360 (Euro-Norm 10025) in sheets with dimensions 1600 x 400 x 5.0 mm, with thickness tolerance ± 0.040 mm on a single sheet.	**Clause D.1**

…

SkyHi must pay Steelmans $25,000 before 1st September 2001.	**Clause C.1**
Steelmans must deliver 10 tons of steel before 1st October 2001.	**Clause C.2**
SkyHi has the right to return the steel within 30 days.	**Clause C.3**
In the event of a return in terms of Clause C.3 above, Steelmans shall refund SkyHi the amount paid.	**Clause C.4**

…

In addition, SkyHi has the following internal organizational policies:

SkyHi Risk Management Procedures

…

Clerks may not buy steel.	**Clause P.1**
Employees older than 25 may buy steel.	**Clause P.2**
Payments of more than $10,000 to suppliers are prohibited.	**Clause P.3**

…

And SkyHi finds itself subject to the following provisions of legislation:

Commercial Trade Act

…

An obligation is fulfilled when all obliged occurrences have happened.	**Clause L.1**
An obligation is violated if it is after the deadline and some obliged occurrences have not happened.	**Clause L.2**
Following successful instigation of the prescribed procedure for claiming compensation, damages for violation of an obligation must be paid, by the liable party, to the party entitled to compensation.	**Clause L.3**

…

SkyHi wishes to store the provisions of their contracts and internal business policies, and the legal regulations to which SkyHi is subject, in a database, so that the provisions can be used to guide the behaviour of their (computer and human-activity) systems. Scripting the system with procedural code is not an option: the sequence of SkyHi's business processes is not static and they do not wish to employ a programmer to sift through and change procedural code to reflect the frequent alterations in contracts, policies, and regulations. SkyHi would like the human and software components in their system to consult the database in order to determine what to do next in the light of a dynamically changing set of provisions. They need to store a history of events and states, so that they know what occurrences have happened over time. They want to see, for instance, what consequential obligations and legal powers resulted, and whether these obligations were fulfilled or powers were exercised. Further, they need to be able to assess whether a given activity is permitted, to determine which historical occurrences were prohibited, and what violations ensued. Finally, they would like to be aware of conflicts and inconsistencies across their various documented policies, and be able to clarify the intended interpretation where necessary.

The application scenario described here has been chosen to include a varied set of realistic legal provisions which we shall scrutinize as we progress:

	See § (pg)
• definitions or 'interpretations of terms' (Clause D.1 defining 'steel', and Clauses L.1 and L.2 defining 'fulfilment' and 'violation'),	5.5 (137) 5.6.1 (142) 5.6.2 (151)
• unconditional obligations (Clauses C.1 and C.2)[1],	5.6.1 (142)
• conditional obligations (the obligation, in Clause C.4, to refund upon return; the secondary obligations to pay damages in the circumstances specified by Clause L.3),	5.6.5 (155)
• legal powers (the right to return the product within 30 days, in Clause C.3; the authority or contractual capacity of an agent to purchase, in Clause P.2),	5.5 (137)
• legal disabilities (Clause P.1),	5.3.2 (132)
• prohibitions (Clauses P.1 and P.3), and	5.3.1 (129)
• permissions (Clause P.2).	5.4 (134)

[1] Technically, these obligations were conditional upon the validity of the contract, but we shall assume the contract is signed and valid, and that these obligations are now unconditional. Subsequent invalidation of the contract may of course void these obligations.

1.3 Research Issues and Motivation

Contracts are central to the operation of commercial organisations in that they constrain and direct the behaviour of a company and its agents. Automated management of business contracts holds the prospect of providing more accurate and efficient commercial operation. We propose to make contracts central to the development and control of e-commerce applications. To this end, we aim to represent contractual provisions in a form suitable for machine interpretation. Contracts may then be queried, executed, and monitored automatically; that is, our goal is to provide contract-driven enforcement. Our contracts are not static but are subject to change and this change must be managed. We propose to develop applications by ensuring that participants adhere to the provisions of the contracts that bind them, and highlighting when this does not take place. This is a novel approach which will integrate the specification of an e-business application with the implementation of its code base.

1.4 Research Goals and Tasks

We take as the input to our process a set of textual business contracts, policies, and regulations, supplied by management, including user requirements documents provided by a business analyst. These define what the various stakeholders (role-players) in the system *can* and *must* do under various circumstances, as well as what the computerized system itself *can* and *must* do under various circumstances. The input documents therefore define the contractual provisions that direct behaviour in both the human and automated systems. Interpreting these documents using automated natural language processing (NLP) techniques is not technologically viable given the current state-of-the-art in NLP [OM96, Pul96, FSS99]. One of our goals therefore is to provide a human analyst with sufficiently detailed methods to guide the interpretation of the specifications and facilitate their input into a computerized system. Our other primary objective is to engineer this computerized system to be

capable of storage, consistency checking, enforcement, and execution of these contractual provisions. Ideally, it should be possible to store the business specification in the database and consult the specification for advice on what to do next.

Managing contracts effectively requires a powerful semantic model and a generic storage framework. If contracts are to be enforced automatically then the representation must capture the relevant semantics in full. Which contract provisions are applicable must be determined in changing circumstances and in the light of frequent alterations and additions to the contracts themselves. The common system development assumption that specifications and contracts are provided once off, up front must be abandoned: the system must modify its behaviour as new directives are added at run-time. Conflict resolution facilities must be capable of highlighting mutually exclusive provisions and deciding between them. Schedulers must enact obligations. Monitors must flag violations. Change management facilities must maintain a history of the nature and status of the organisation's past contracts in order to resolve any disputes that may arise.

Following our detailed review of the adequacy of related work in Chapter 2, we conclude that chapter with a more specific list of requirements for a contract-driven e-commerce application development environment.

The implementation context for the desired environment is as shown in Figure 1 below. Business analysts feed the business policies defined in the business's contracts, policies, and regulations into the system. The business policies are stored as structured data. Fulfilment occurrences, environmental happenings, and further contracting occurrences are added to the occurrence store through defined interfaces and checked against existing contracts by an active wrapper. Occurrences are triggered automatically by the system in accordance with the policies defined in the contracts (specifications) in the occurrence store.

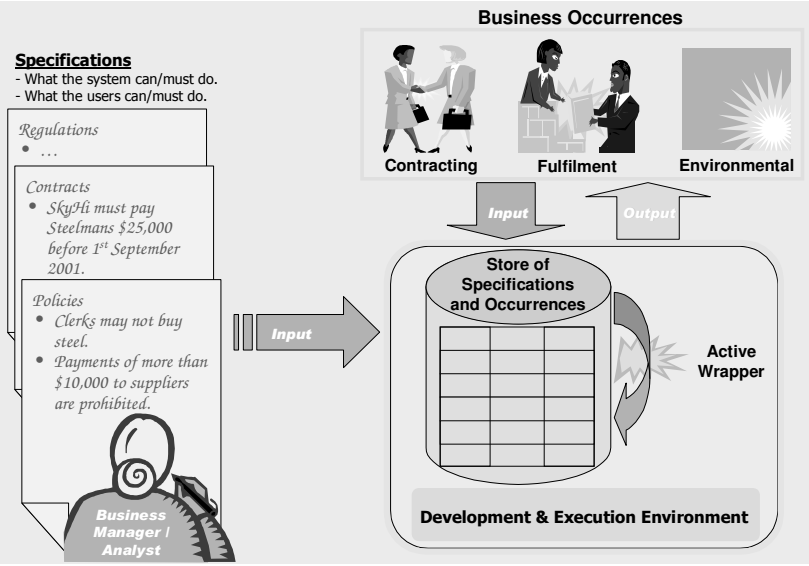


Figure 1: Contextual overview of desired solution

1.5 Contributions

We provide a set of guidance rules, which can be employed by an analyst to expose salient occurrences in English language user requirements documents, such as business contracts, policies, and legislation [AB2000, AB2001a]. Our investigations into event semantics have led to the definition of a database schema for the representation of these variable-attribute occurrences [AB2001b], paving the way to interrogation and execution of stored e-commerce application specifications. Our EDEE prototype provides a platform-independent active wrapper [AB2001c], which allows us to record, reason about, and enact contractual provisions [AB2001d, AB2002a]. We demonstrate a novel query storage and coverage determination mechanism, which allows contract performance monitoring and facilitates dynamic consistency checking of contracts against policies [AEB2002a, AEB2002b]. A new model of the life and times of identified and situated norm instances is proposed [AB2002b, AK2002]. The model is used in our contract-driven and legislation-aware workflow automation approach, to support conflict resolution [AEB2002c].

The work described here contributes to the understanding of the representation and implementation of contracts, policies, and legal requirements, for business process automation. The use of a database to record history in an integrated form enables querying of state and history, which is essential to the abstract modelling of processes at the business level. We have proposed a significant attempt to advance the state of the art in the capturing of user requirements and their mapping to computer systems functions such as access control, enactment, and audit. This research may stimulate new approaches to delivering security and integrity to business processes, and has the potential to feed into developer tools based on policy specification and enforcement. The novel development style described could have a significant impact on the automation of activities in commercial organisations and would be an important contribution to workflow management in e-business.

Progress towards the creation of an integrated methodology and environment for provision identification, representation, storage, conflict detection and resolution, monitoring, and enforcement presents a significant engineering challenge, and constitutes the main contribution of this thesis.

1.6 Dissertation Outline

This dissertation is organised as follows:

Chapter 2 investigates background and related work, providing a broad review of treatments of events, rules, policies, executable specifications, and contracts by previous authors. The chapter highlights the weaknesses in conventional approaches to specifying and animating business workflows. Based on the shortcomings of these approaches, and informed by some suggestions of deontic logic, it derives requirements for a methodology and infrastructure for developing and executing electronic commerce applications.

We begin to tackle the first of the identified requirements in Chapter 3, which introduces a representation of business occurrences – that is, events and states – which is grounded in philosophy and is semantically rich enough to record both

workflow and association occurrences in e-business. We show how descriptions of occurrences and entities are stored using identified queries, and we demonstrate a novel incremental detection mechanism, which makes use of partial re-evaluation of stored queries to determine whether an item is covered by a stored query.

Motivated by the requirement for improved consistency between the analysis and implementation phases of e-commerce application development, Chapter 4 details a set of guidelines that may be employed to expose salient occurrences in an English-language specification. These aid the process of refining analysis documents, such as user requirements specifications and business contracts, into structured provisions that can be stored in a database.

Chapter 5 focuses on provisions, explaining the types of provisions, and their storage. The chapter demonstrates that contract-related occurrences can be recorded within the data schema used for recording operational workflow occurrences and associations.

Mechanisms for conflict detection and resolution are explored in Chapter 6.

Monitoring and enforcement techniques for a contract-driven e-commerce application execution environment are presented in Chapter 7. The chapter also outlines the prototype implementation, EDEE, of the techniques introduced in this thesis.

Chapter 8 discusses the solutions presented in this dissertation. It reviews the strengths and weaknesses of the approach, gives a detailed experimental evaluation, and suggests directions for future research.

Chapter 9 concludes this dissertation, and highlights the main contributions.

Chapter 2

Analysis of Related Work

We saw in the previous chapter that traditional business process application development approaches suffer from a *sequential invocation paradigm* that is *opaque* and lacks an explicit representation of the norms governing behaviour. In particular, the *changing rights and duties* of parties in contracts are typically overlooked. In this chapter, we explore and criticise a broad range of technologies and formalisms that have investigated some of these issues.

A sequential invocation paradigm, while efficient, does not allow us to monitor for the occurrence of events and states against a changing set of contractual provisions. The problem is two-fold: *representing* events, and *detecting* events. Section 2.1 introduces the technical literature on event monitoring, which tackles the second aspect, and includes a brief review of the philosophical literature on event representation, which provides some insights into the first.

Several research initiatives exploit an event-driven execution paradigm and attempt to make business rules, policies, and processes explicit and executable. Section 2.2 examines the rule-based approach, while Section 2.3 surveys policy systems for access control and network and system management. Techniques for business process modelling and animation, such as workflow process definition languages and executable specifications, are examined in Section 2.4.

Section 2.5 evaluates the notion of 'contracts' used in a variety of implementation technologies, and explains how these metaphorical borrowings of the term differ from business contracts, which make rights and duties explicit. Finally, we look at the literature on formal deontic logic (Section 2.6), which has been employed for representing and reasoning about obligations and permissions in moral law, and we contrast these with the obligations and permissions used in business law.

In each subsection, we highlight the major current approaches and identify pressing issues and detailed requirements. Based on the gaps in existing approaches, the chapter concludes with a list of high-level requirements for a comprehensive e-commerce application development framework that expresses and exploits structured abstractions of business occurrences and legal provisions. These requirements are then systematically tackled in the remainder of the thesis.

2.1 Event Systems

Event monitoring is typically employed in three major styles: publish-subscribe event notification (comparable to continuous query), active database triggers, and system event schedulers.

Publish-Subscribe Event Notification and Continuous Query

Publish-subscribe event monitoring services, such as **CEA** [BMBH2000, BHMM2001, BHMM2002], **GEM** [MS97], **Siena** [CRW98], **Eve** [GT98, Tom99], **JEDI** [CDNF2001], **DERPA** [CDDF98], **Elvin** [SABH2000], and **Le Subscribe** [FJLP2001], focus on event detection and filtering. In publish-subscribe, services advertise (*publish*) the events they can notify, consumers *subscribe* to event patterns, and event brokers *notify* interested consumers upon receiving events that match a pattern that a consumer subscribes to. Event-monitoring services allow composite or aggregate events to be fired upon recognition of patterns of primitive events.

Continuous queries are standing queries that monitor source data and notify the user whenever new data matches the query [LPT99]. **Tapestry** [TGNO92], a

purpose-built application designed for filtering mail and news messages, uses append-only tables and notifies users when data matches a stored description. **OpenCQ** [LPT99] generates database-dependent trigger scripts for the situations to be monitored, and uploads each of them to the relevant database; currently only Oracle is supported. **NiagaraCQ** [CDTW2000] defines continuous XML Query Language (XML-QL) queries on XML files, storing delta files with recent changes, and notifying relevant XML document revisions to interested parties.

Publish-subscribe and continuous query mechanisms are intended for targeted dissemination of new, topical information to users. They have been used for detecting events from remote sensors, or informing interested parties of changes to a database or document. Composite event languages primarily support event detection and notification distribution and have weak support for imperative aspects - GEM is an exception to this and provides basic action facilities.

Active Database Triggers (Event-Condition-Action rules)

Active databases employ an **Event-Condition-Action (ECA)** paradigm: on the occurrence of a certain event, if the condition is met, then an action is fired. Event-Condition-Action rules, or **triggers**, are widely implemented in active databases [PD99].

A variety of mainstream commercial relational databases, such as **Microsoft SQL Server**, **Oracle**, and **Sybase**, implement triggers based on the **SQL/99** [ISO99a] or previous SQL standards. SQL/99 triggers are limited to monitoring simple `UPDATE`, `DELETE`, `INSERT`, and (sometimes) `SELECT` operations on single tables.

Academic active database projects include relational active databases such as the **Postgres Rule System** [SHP88, SK91], **Ariel** [Han92, HBC97], **Datex** [BM93], and **Starburst** [Wid96], and object-oriented active databases such as **SAMOS** [GD93, GD94, DFGG2000], **Ode** [GJS92a, GJS92b], **Sentinel** and **Snoop** [CKAK93, CM94a], **NAOS** [CCS94], and **TriGS** [KRRV94, Ret98, KRRS2001].

As an example of a trigger specification, Starburst allows the creation of rules through the execution of statements of the form:

```
CREATE RULE rule_name ON table WHEN INSERTED/DELETED/UPDATED
[IF condition] THEN action_list
PRECEDES rule_list FOLLOWS rule_list
```

System Event Schedulers (Event-Action Systems)

An abundance of commercial system event schedulers exist: industrial products include **Unix's cron**, **Microsoft's Windows Tasks Scheduler**, **SRO Event Manager**, and **Unisyn Automate**. In the academic realm, a prototypical scheduler is the **Yeast** event-action system [KR95]. Upon detection of system-level events, such as windows opening, applications closing, or timer events, actions are invoked. Actions involve running a command line script or making a method call to a system application programming interface (API): for example, backup or archive files, download emails, run a virus scan, or pop up a message window.

Critique

Traditional event systems present us with a number of issues:

2.1.1 Expressiveness of event representation

The literature on events can be broadly divided into *philosophical* treatments, which explore the representation and logic of events, and *technical* approaches, which look at efficient event monitoring. The divisions between logicians and systems programmers in the approaches adopted for the representation and application of event abstractions are marked.

Philosophical Issues: Representation of Events

The notion of individuated occurrences was arguably pioneered by the philosopher Donald Davidson [MDBS69, Dav80] who suggested that events such as 'the *stabbing* of Caesar by Brutus' are identified particulars. The basic argument is that the traditional predicate logic representation

stabbing(Caesar,Brutus)

ought to individuate the occurrences of stabbing so that separate stabbings may be distinguished. Davidson's original proposals [Dav80, p118], would then yield:

$\exists x$ *stabbing(Caesar, Brutus, x)*

where *x* denotes the particular occurrence of stabbing being referred to. An active philosophical debate has ensued for decades. A variety of authors – see, for example Mourelatos [Mou78], Bach [Bac86], Bennett [Ben88], Verkuyl [Ver96], and Steedman [Ste2000] – have looked at philosophical nuances and shared abstractions for events, states, and processes. Contributors who have studied representational issues have included Parsons [Par90]; Jurafsky and Martin [JM2000]; Higginbotham, Pianesi, and Varzi [HPV2000]; and Kimbrough [Kim98a, Kim98b, Kim2001]. Parsons puts forth a more granular refinement of Davidson's representation, motivated by, amongst other reasons, the need to straightforwardly infer 'there was a stabbing', 'Caesar was stabbed' and 'Brutus stabbed' from 'Brutus stabbed Caesar', without requiring additional meaning postulates. Parsons' version becomes:

$\exists x$ *stabbing(x)* $\wedge$ *agent(x, Caesar)* $\wedge$ *theme(x, Brutus)*

Kimbrough, in his **ESΘ theory** [Kim98b], adopts the Parsonian view. Both propose that the quantifier variable (*x*) may individuate instances of any type of eventuality – what we term here **occurrences** – including events, processes, and states. Kowalski and Sergot [KS86] provide perhaps the seminal formal treatment of instantaneous events in a logical calculus, showing how events initiate and terminate prolonged properties. This work has been extended by Sadri and Kowalski [SK95] and others. Daskalopulu [Das99] suggests that Kowalski and Sergot's work is compatible with Kimbrough's approach to eventuality individuation.

With the notable exception of Kimbrough – who speaks of events such as *ordering*, *promising*, *invoicing*, *delivering*, *confirming*, and *debiting*[2] – other accounts of event semantics restrict themselves to philosophically interesting, but commercially less pertinent events, such as *stabbing* and *killing*. Furthermore, Kimbrough in his **Disquotation Theory** [Kim2001], also in collaboration with Moore in their **Formal**

[2] As per Parsons' suggestion [Par90], we use the '–ing' form of verbs as their canonical form.

Language for Business Communication (FLBC) [KM97, Moo2000], extends the simple sentences treated by other authors with sentences containing embedded propositional content. Such sentences include '*promising* that …', '*requesting* that …', and other examples prevalent in business communications. In each case, propositional content (e.g. 'I pay') is bracketed or 'quoted' within a sentence that indicates attitude[3] (e.g. 'I promise that [I pay]'). Kimbrough and Moore's contribution is in formalizing the suggestions of speech act theorists [Aus76, Sea69, SV85], and in applying these ideas to deontic reasoning [Kim2001] and business messaging [KM97, Moo2000]. Daskalopulu and Sergot [DS2002] investigate the computational aspects of Kimbrough's emerging theory, showing a logical programming implementation, in Prolog, of some of his proposals.

Valuable strides have been made towards practical application of event semantics and speech act theory in commercial messaging and inferencing. The foundational work demonstrates the promise of applying the rich semantic view of events and states from the philosophical literature to real world systems. However, specific software architectures and services for the creation and storage of large numbers of user-supplied event matching patterns, and event detection, lie outside the scope of philosophical concern. For this we turn back to the technical literature on event monitoring, which adopts a more impoverished view of events, but treats algorithms and data structures to support the practical implementation of event monitoring in more detail.

Technical Implementation: Event Representation

Technical approaches to event monitoring are typified by representations of events and states that differ vastly from the recommendations of the philosophical literature. Koch's **Event Definition Language (EDL)** [Koc97] provides perhaps the most pointed evidence of the primitive representation of events sometimes used. EDL takes a limited size buffer of integer values and notifies a named event each time a threshold is reached by the most recent value or by the running mean or median of values in the buffer.

[3] Austin [Aus76] and Searle [Sea69] use the term 'illocutionary force' rather than 'attitude'.

Problematically, business events are at a higher level than the data management events used in database triggers [PS98]. Database triggers monitor only for database-level events, not business occurrences. Semantically richer *named, typed, and parameterized events* are provided in some approaches [BMBH2000, Spi2000]. Notably, however, event representation is not guided by philosophical recommendations. Parameter choice and naming is arbitrary, and not driven by analysis of *participants and their roles* in the event, as is recommended in the artificial intelligence and natural language representation literature [All95, Gri94, JM2000, Par90, Sow2000]. Most event languages surveyed, including composite event languages and Event-Condition-Action rules, treat events as instantaneous [Bac96, Spi2000, PD99] although GEM [MS97] treats composite events as durational (having both start- and end- events). Various authors in the fields on logic and linguistics [Bac86, Hay95, Kim2001, KR93, Mou78, Par90, Sow2000] contend that it is natural to consider some events (or, rather, 'eventualities') as occupying intervals and being durational. In contrast to these philosophical views, systems programmers would typically regard durational events as states. An integrated stored history of both *instantaneous events and prolonged states* is essential for recording, monitoring, and auditing business processes.

Requirement 1	**The information model must be able to describe occurrences of past events, states, and processes, as well as the (active and passive) role-players involved.**

While typical workflow events like 'withdraw', 'deposit', and 'pay' are unproblematic for the representation of events used in much of the event monitoring literature, certain types of events cannot be defined. Speech act events *with nested propositional content*, such as those expounded by Kimbrough (op cit, page 17), are not expressible. Events of this type include occurrences of promising, authorizing, forbidding, requesting, expecting, attempting, or planning, which are common in commerce.

Requirement 2	Occurrences with nested propositional content must be expressible.

2.1.2 Monitoring for recent, non-persistent occurrences

While efficient for real-time monitoring, event monitors do not support long-term event storage and querying which is essential for e-commerce. They do not explore the monitoring and assessment of events against a changing set of stored provisions, nor the logically derivable implications of events in terms of user-defined policies. In the absence of integration with a business policy system, event monitoring systems cannot check which business policies are applicable in the current circumstances. Though CEA has been integrated with the OASIS access control policy language (page 34) and GEM has been integrated with the Ponder network management policy language (page 36), these are systems-level approaches unsuited to the monitoring of high-level business contracts.

The event detection mechanisms employed by publish-subscribe and active database systems (summarized in Table 1 below) match against a rapid stream of current events and are optimized for high-performance with transient events and conflict-free specifications.

Detection Mechanism	Implementation
Regular Expressions / Finite State Automata	JEDI, Ode, DERPA
Petri Nets	SAMOS
Trees / Graphs	CEA, GEM, Sentinel, Eve, Snoop, Schwiderski
Row Markers (Locks)	Postgres
Multi-dimensional Indexing	Le Subscribe

Table 1: Event detection mechanisms used in publish/subscribe and active databases

Regular expressions and **Finite State Automata** recognize events when a Finite State Machine reaches its final state. They have no memory of which particular transitions have been followed to arrive at that state. As counts of events cannot be modelled with pure automata, Ode's implementers extend the automata with vectors of count variables. They warn, though, of a combinatorial explosion in space complexity when trying to store all information derivable from the transition history of a Finite State Automaton. **Petri Nets** generalize Finite State Automata, making recent transition history accessible. Some history for the current rule firing is held in tokens, but more distant past events are not stored.

In **tree** and **graph-based** detection approaches, each node in the graph maintains an event buffer, with events moved from the child node's buffer to the parent node upon event detection, or discarded if the current node is a root node. Most approaches use one or more of four hard-coded event consumption policies – Recent, Chronicle, Cumulative, or Continuous [CM94a, PD99]. These consumption policies choose which of a time-limited set of buffered events to use in matching against a given subscription. While a given event may be sent to more than one tree instance at the time of matching, events already matched are not available later to other rules for firing and no event history is stored.

Row-markers (record- and relation-level locks) monitor for simple events: if a marked row is altered, inserted, or deleted, the trigger is fired. **Multi-dimensional indexing** approaches provide rapid performance when the number of subscriptions is large, as they employ indexing to look up which matching patterns pertain to a single, current event.

Matching against transient, consumable events is ill-suited to commercial applications where event consumption is determined by business policy which assesses event salience over time. The fact that an event has been matched against an expression mentioned in a provision does not inhibit it from matching against further expressions, and seldom means that it can be discarded. A chronicle of the history of events is important for audit and evaluation of performance against contract, for

the gathering of performance metrics for process improvement, and for retrospective evaluation and justification of actions. The ability to determine which events brought about which obligations, is critical. Equally essential is the ability for a historical event to bring about a new obligation even if it has previously brought about other obligations – matched events therefore cannot be summarily garbage-collected. Events must persist because contractual outcomes may depend upon events in the distant past.

Requirement 3	**Matching (occurrence detection) must be against a long history.**

It is not sufficient to merely provide a record of method invocation history, as these are low-level service requests. An occurrence history must store a history of business events and states in order to be assessable against a business contract.

Requirement 4	**Persistent storage of business-level occurrences is required.**

Event-detection trees, graphs, and automata hard-code access paths to the events: the detection graph would need to be traversed to find events; this is further complicated by the short retention span of events. Occurrences cannot be looked up via ad-hoc queries. This is not well suited to electronic commerce applications, which require a persistent and easily interrogable record of occurrences. The **Herald** event store for multimedia applications [Spi2000] demonstrated the viability of matching subscriptions against events in persistent storage. However, Herald's implementation is optimized for a single access path to events: the time-ordered access used in multimedia replay. CEA (page 14) provides both ODL- and XML-based event stores queried via OQL and XPath respectively. A composite event retains all its components, each with a timestamp. Event audit can be handled by a background write to an audit store. The intended applications of CEA are mobile

entity tracking, controlling interactive multimedia applications, and monitoring security conditions.

Requirement 5	Business applications require occurrences to be accessible via ad-hoc queries, rather than via fixed access paths.

Patankar and Segev's **Business Event Manager** [PS98] offers services such as persistence and sharing which are not provided by existing active database systems. They attempt to store a history of business events using an irregular-time-series data type proprietary to **Illustra/Informix Universal Server**. Each event type is stored in its own time series, indexed by occurrence time. Simultaneous events are not supported as events are identified by occurrence time rather than by an occurrence identifier. Though they see the need to conceptually support event attributes of any data type, their implementation in Illustra restricts event attributes to numeric types only, severely limiting the expressiveness of their events. Patankar and Segev do not specify an event detection algorithm, and propose only an event scheduling approach whereby scheduled occurrences are automatically inserted into a future event queue. Their approach is therefore similar to system event schedulers (page 16), except targeted at scheduling business events rather than system administration tasks, and storing the event log in a temporal database rather than in a textual log.

2.1.3 Focus on system administration, not business process automation

System event schedulers (page 16) are well-suited to automating system administration tasks, but inappropriate for workflow automation. While these products provide event detection facilities, they can monitor only a limited set of system-level events such as file modification, drops in available file-system capacity below a certain threshold, or user login. A typical system-level rule in an event scheduler might say:

> Effective from 13 January 2003 to 13 March 2003, print accounting reports before 8am on the last working day of every month

A typical business-level rule appears superficially similar:

> Effective from 13 January 2003, customers must settle their bills before 5pm on the last working day of every month.

In the case of a system-level rule, the rule can only ever be a rule to be automated by the system; the rule may not mention business objects (such as customers) to be monitored. Indeed system event schedulers provide no general data model. The system-level rule is not subject to override by other rules, and conflicts between rules are not detectable. The system-level rule can be disabled by another rule, but only if it is specified by identifier – it cannot be selected by attributes such as the identity or organizational position of the rule's specifier, the rule's location in a document, or its time of specification. The business-level rule may be voided for a variety of reasons: because it is overridden by a conflicting rule from a higher authority, because it is contained in a section of a contract declared null and void, or because it was written before a certain date. Failure of the system-level rule usually initiates an e-mail notification to the system administrator. A violation of a business level rule may result in liabilities, such as obligations to pay interest on overdue bills.

Requirement 6	**A business-level, rather than technical-system-level, approach is needed for business process automation.**

2.2 Rule-based Approaches

Business rule implementations vary from integrity constraints and procedural rules, to logic programs, expert systems, and rule engines. Event-Condition-Action rules, which we introduced earlier (page 15), may also be considered a rule-based approach.

Integrity Constraints

Integrity constraints, assertions, and pre- and post-conditions, are simple validity checking conditions overlaid onto existing procedural approaches.

SQL/99 [ISO99a] **integrity constraints** and **assertions**, implemented using `CHECK` and `ASSERTION` statements, are a rudimentary form of rule that describe conditions that a row, column, table, or database must satisfy [Dat2000]. Insertions, deletions, or updates that violate these hard constraints are rejected.

For designers in the object-oriented community, integrity constraints, pre- and post-conditions, class invariants, and assertions are semi-formally expressed using the **Object Constraint Language (OCL)** modelling notation [OMG2001]. Integrity constraints are operationalized in the **Eiffel** language [Mey99] or **iContract** pre-processor for Java [Kra98], where conditions are tested on method entry or exit.

The logical language **KAOS** [Lam2001] allows the specification of constraints on domain objects (similar to class invariants), and constraints on processes or requirements (similar to necessary and sufficient pre- and post-conditions). KAOS does not describe how conditions are monitored, as the approach downplays operationalization and concerns itself with formal reasoning about whether goals are achieved, maintained, or avoided, using temporal logic primitives (*SomeTimeInFuture*, *AlwaysInFuture* ... *Unless* ..., and *NeverInFuture*).

Procedural Rules

Procedural approaches embed activity-triggering rules. For instance, internet application servers like **ATG Dynamo** [ATG2001], **BEA Weblogic** [BEA2001], **Microsoft Site Server 3.0** [Mic2001a] and **Commerce Server 2000** [Mic2001b]

implement simple procedural rules for content personalization. These are typically encoded as sequential `if … then …` statements in VBScript, JavaScript, or Java, and embedded in web-page generation templates.

Logic programs, expert systems, and rule engines

Full-fledged, non-procedural rule-based implementations include logic programs, expert systems, and rule engines.

Prolog is a declarative, logic-programming language targeted at inferencing applications. It employs a depth-first, left-to-right unification algorithm to infer conclusions from rules (patterns) applied to facts or assumptions (objects) [CM94b]. Rules are expressed as Horn clauses, a subset of First Order Logic. Negation by failure is employed.

Situated Courteous Logic Programs (SCLPs) [GLC99, IBM2001, RGW2002] extend logic programs with labelled rules (for conflict resolution) and procedural sensors and effectors.

Expert systems attempt to match a set of facts to patterns defined in unordered `if … then …` rules. Forgy's **RETE** algorithm [For82] is widely used in expert systems shells, for rule condition testing. RETE is common in **OPS5** [For82] descendants including NASA's **CLIPS** [Ril2001], and the Java implementation of CLIPS, Sandia's **JESS** [FH2001]. It is also embedded in commercial rules engines such as **Blaze and Brokat Software's Advisor** [Bla2000], **Versata Logic Server** [Ver2001], **Usoft** [Uso2001], **ILOG Rules** [ILOG2001], and **Haley's Eclipse** and **Café Rete** [Hal2001]. Some extensions may be provided. For instance, ILOG extends RETE with a notion of rule packets or rule groups. These allow the activation and deactivation of sets of rules.

RETE only tests rules that may have been affected by the most recent change in data, thereby making it significantly faster than the naïve approach of running data through the entire set of `if … then …` rules in a procedural manner. Further, RETE's constraint checking approach avoids the explicit invocation problem of SQL triggers, Java, and other procedural approaches, which leave programmers with the difficult

and error-prone task of inserting checks in all places where constraints might be violated [Bla2001].

In RETE, new facts flow through a dataflow network of one-input (α) and two-input (β or join) nodes, which test whether conditions (i.e. constraints or queries) are met. Facts are stored at β-node memories to avoid the need to re-evaluate facts already matched. Facts that reach terminal nodes activate rules, which are added to an agenda. Optimization techniques that may be employed include sharing common sub-conditions between rules [FH2001].

RETE is suitable only for small main memory data sets as the large number of facts stored at β-node memories lead to exponential worst case storage requirements: a storage cost of $O(t^j)$ for *t* facts in a database, and a single condition with *j* joins [Mir87, BGLM91]. Friedman-Hill [FH2001] explores a simple example where the compiled RETE data-flow network for a single rule with 5 conjuncts (4 joins), when applied to 10 facts (events), stores 10,000 partial matches at the last join node. JESS provides a profiling command that allows developers to restate the order of conjuncts in the rule, so that they may decrease storage overhead by placing the most selective predicates first. However, it seems unreasonable to expect developers to manually profile and then restate and recompile their rules into a new RETE network each time the selectivity of predicates changes. A variant of RETE, **TREAT** [Mir97], stores predicates (rather than facts) at β-nodes, in an effort to trade off search time against memory overhead. However, the structure of the compiled network in TREAT is again dependent on the order of statement of conjuncts in the rules, rather than on the selectivity of predicates, meaning that the algorithm performs poorly for certain data profiles. **Gator** [HKNPV98], the trigger mechanism used in Ariel (page 15), performs heuristics-based optimization of RETE and TREAT networks, to improve match efficiency based on data profiles.

Critique

In the context of business process enactment, the following primary concerns pertain to rule-based approaches:

2.2.1 Appropriateness for event monitoring

Expert systems are ill-suited to monitoring applications [Ril2001]. Firstly, reasoning about events over time is not a built-in feature. Riley [Ril2001] recommends that an additional slot be added to facts to store their creation time. Rules which reason based on the contents of this slot must then be added. However, as occurrence time does not uniquely identify simultaneous occurrences, this approach is flawed, and use of an occurrence identifier, as suggested by the philosophical event literature, seems preferable. Secondly, the algorithms assume that data changes slowly over time. Riley suggests that rapidly changing data may be partially addressed through creation of a bespoke pre-processor. Two alternatives are proposed. If the change in sensor value is insignificant – for example a change from 10 to 10.1 – the pre-processor avoids the retraction of the old fact and the assertion of the new fact with the changed sensor value until the change is convenient. An alternative is to convert numeric values to symbolic values such as `low`, `nominal`, and `high` and retract the old fact and assert a new one only if the symbolic value has changed. Riley remarks that a major drawback of this approach is that C code has to be written to pre-process the data.

2.2.2 Management and control of small, static rule sets

Expert systems which compile rules into RETE networks are not intended for large, dynamically changing rule-sets: the typical expert system assumes a fixed set of rules, compiled into a static constraint checking network, and a variable set of facts [FH2001].

Requirement 7	**Situations should be interpreted against a dynamic set of rules.**

Rule engines are primarily directed at inferencing, with rule management being unattended to. For instance, **IBM CommonRules** [IBM2001] stores rules in text files, making it difficult to search for and view rules. RETE and TREAT are

restricted to in-memory rule sets, and assume that rules are read from text files; manageability of large volumes of rules is therefore an issue. Similar concerns affect active databases: triggers are stored in text files and compiled; the rules are not stored in interrogable form in a database. Prolog rules are unlabelled, and held in text files, though some implementations allow dynamic insertion of new rules into a data store.

Lee and Ryu [LR94] comment that, rather than modifying rules using an ordinary text editor, updates should pass through deontic controls. The policy-based system **Ponder** [DDLS2001] goes some way towards preventing undesirable updates to rules through the use of meta-policies [LS99].

Commercial rules repositories, like **BRS RuleTrack** [BRS2001], can be used for recording and organizing rules. RuleTrack is limited to storing textual descriptions of the rules, which are not machine-interpretable.

Requirement 8	**A large and growing number of machine-enforceable rules should be controlled and managed in a database, not haphazardly distributed in text files.**

2.2.3 Absence of native conflict detection

In classical rule-based approaches, the semantic form of rules is inaccessible; rules cannot be queried and conflict detection based on properties of rules cannot be performed. Typically, no facilities are provided for ascertaining when two rules might compete, making it difficult for the system administrator to determine when the introduction of a new rule would impact upon the firing of rules lower down in the rule file or in other rule files. In environments where rules are specified by distributed and semi-autonomous authorities, conflicts frequently arise. Where thousands of rules exists, it is not feasible to expect such conflicts to be located by humans.

Requirement 9	**Analytic conflict detection is desirable.**

Importantly, RETE, TREAT, and Gator can only check whether fixed-value tuples match patterns (queries), and cannot determine whether the results of some patterns (queries) are subsets of the results of others. Similarly, Prolog provides no native facilities for determining overlaps between rule conditions, as its intention is to match facts to rules, rather than to compare rules to each other. Current rule based approaches focus predominantly on *object-pattern matching*, rather than *pattern-pattern matching*, and consequently do not provide native conflict detection facilities. The RETE approach generates and then compiles code to represent α and β nodes, and then feeds the data through these generated classes. Constraints therefore cannot be individually identified and their semantic structure cannot be queried as they are split into the compiled code of a data flow network, which is opaque. Ability to interrogate the semantic structure of the rule set would be necessary for pattern-pattern matching and analytic conflict detection.

Requirement 10	**Pattern-pattern matching functionality is required for analytic policy conflict detection.**

2.2.4 Priority-based conflict resolution

Rule conflicts are not well dealt with in expert systems, where, as for active database systems, resolution is via implicit ordering or explicit priority numbering.

Conflict resolution in active databases is usually limited to hard-coded rule ordering or numeric prioritization. Prioritization is difficult to maintain when the rule set is evolving. As is evident from its `PRECEDES` and `FOLLOWS` clauses (page 16), rule ordering (conflict resolution) in Starburst is hard-coded through relative priorities specified in the rule definition, and cannot be driven by external meta-policy. NAOS also employs `PRECEDES` clauses, while SAMOS uses `PRIORITIES BEFORE|AFTER rule_name` for similar effect. Ariel, Postgres, and TriGS require an absolute priority level for each rule. For instance, in TriGS, the priority information of each rule provides absolute execution priorities in terms of number (`0=highest` priority, `1=medium` priority,

etc.). Static linear prioritization through implicit textual ordering or priority numbering is not feasible for large rule sets, especially as rule precedence is often context-specific [Mak88]. JESS [FH2001] associates a `salience` property with each rule, proceeding in a manner similar to that employed in TriGS. Rules with the highest salience are fired first, followed by those with lower salience. Salience values can be specified using constant integers, variables, or function calls. For rules with the same salience, a conflict resolution strategy may be written in Java. The Java code traverses the list of activated rules in the agenda and fires rules in chosen order. Though provided as a feature, use of salience is discouraged as causing rules to fire in a certain order is considered bad style in rule-based programming.

In Prolog, conflict resolution is achieved by implicit textual rule ordering in combination with deft application of the `cut` command to control backtracking (the search of alternative solutions). Rules are unlabelled, and, as in active database and expert system approaches, their semantic form and attributes (e.g. author, specification time, jurisdiction, author's current and previous roles, etc.) are inaccessible, so cannot be used for the selection of rules to regard as void.

SCLPs employ an `overrides` predicate to specify priorities between labelled rules. For example, rules may be tagged as `medium`, `high`, `very_low`, or `lowest` priority, or markers such as `amazon` and `ebay` may group them.

In Microsoft Site Server, conflicts are resolved by firing only the first applicable rule. As in other procedural approaches, this is achieved by nested `if…then…else…` control-flow statements organized in textual order. Similarly, `switch…case…` control structures, as available in C and Java, use `break` statements to prevent other conditions from being evaluated.

In KAOS, goals are named, and goal priority is mentioned as a means for resolving conflict between goals (goal weakening or sacrificing) but is not included in the formal representation.

None of the traditional rule languages or goal-driven approaches natively formalize the essential primitives of commercial law that pervade e-commerce. According to the jurisprudential theorist Wesley Newcomb Hohfeld [Hoh78],

notions such as duties, privileges, powers, and immunities, are among the fundamental legal conceptions, and are essential constructs for representing legal relations between parties in contracts.

The constraint-checking approaches of RETE and TREAT do not emulate legal reasoning. Constraints are taken as hard, with no facility for selective voidance of certain constraints under special circumstances, as is typical in judgement formation. During the assessment of events against legal provisions in legal reasoning, the relaxation (voidance) of certain constraints may be justified by defined fairness principles that apply in the case of conflict between constraints.

SQL/99 constraints and assertions are taken as absolute and strictly inviolable, with no facility for voidance under special circumstances. That is, a rigid system-wide conflict resolution strategy is imposed, where prohibitions are always taken to override permissions and obligations. Rejection of updates regiments the enforcement of prohibitions, and does not allow for the common case in commerce where prohibitions may be violated at some cost.

Requirement 11	**Fundamental legal conceptions – such as duties, privileges, powers, and immunities [Hoh78] – must be natively incorporated in the development approach.**
Requirement 12	**Rule attributes – such as author, specification time and document or utterance location, scope, and jurisdiction – should be stored, as should attributes of entities related to rules – such as author's roles over time.**
Requirement 13	**Reasoning techniques should emulate legal reasoning. Conflict resolution facilities should allow selection of applicable rules based on recency, specificity, location, authority, or other criteria [GLC99].**
Requirement 14	**Context-specific rule precedence must be supported.**

2.2.5 Synchronous invocation style

In business, events trigger obligations, permissions, and construals and these motivate actions; whereas in the ECA rule paradigm events directly trigger actions. In Prolog, inferencing may be supplemented with procedural side effects: that is, invocation of operations during clause (goal) evaluation. These synchronous invocation styles are inappropriate in e-commerce contracts where conditions create obligations rather than directly invoke actions. An asynchronous 'Event-Condition-Obligation' paradigm would appear to be more appropriate. In e-commerce the existence of a condition does not imply that all obligations arising from that condition will necessarily be fulfilled, as is suggested by ECA rules. Rather, it implies that such obligations *exist*; subsequent obligation choice decides which of the prima facie obligations apply, and action choice decides which, of those that apply, will be fulfilled. ECA rules and procedural side effects assume that managed agents lack free will and always act in a certain manner upon occurrence of triggering events. This assumption is unreasonable in commercial environments where free agents and unpredictable environmental forces may lead to deviation from prescribed norms [DDM2001]. ECA rules therefore require an indirection whereby conditions trigger the creation of obligations. This makes it possible to reason about which of a set of conflicting obligations to void, violate, or fulfil. 'According to Clause C.1, SkyHi is obliged to pay Steelmans' does not mean that SkyHi is authoritatively obliged to pay Steelmans, only that some clause says it is obliged. A government-imposed steel embargo may void the obligation to pay Steelmans, were Steelmans to be domiciled in a blacklisted country. A set of norms must be followed in choosing which clauses are nullified and which prevail in the circumstance. It is only once an obligation or construal has been chosen in accordance with these principles, that the obligation should be enacted by triggering suitable action, or that the construal should be accepted as a basis for decision-making.

Requirement 15	**The treatment of obligations – whether to void or violate them in a given case – must be user-definable.**

A further incompatibility between business contract fulfilment and the synchronous procedure invocation approach of rule-based systems is that obligations typically specify deadlines, giving some leeway within which actions must be initiated. Deadlines provide the flexibility to initiate actions at a more optimal time, rather than immediately invoking operations the moment a condition occurs (as happens in ECA triggers) or the moment a clause is evaluated (as happens in logic programs like Prolog). Immediate invocation may be sub-optimal when the costs of overly-hasty action exceed the benefits of waiting for a more opportune time. A business-driven paradigm should therefore differ from the traditional ECA approach in that it is obligations (and other legal relations) that come about on the occurrence of events, and actions are asynchronously brought about by components which determine which obligations still need to be fulfilled, and whether to fulfil them now or later.

Requirement 16	**Asynchronous fulfilment of chosen obligations must be supported as an enforcement style.**

2.3 Policy-based Systems

Policy-based systems extend from action-constraining *access control* architectures, to action-initiation frameworks for *network and system management*.

Access Control Policies

Access control architectures express and implement authorization policies. Here we sample a few representative implementations.

OASIS [BMBH2000] manages, in a distributed setting, credentials that a principal must hold in order to enter a role and access a service or resource.

LaSCO [HPL99] is a language for stating and enforcing authorization policies for Java programs. LaSCO provides a policy compiler that adds policy checks into Java source code as wrappers on method invocations. LaSCO can express only what

events must be present, but not what events must be absent, for the policy to be applicable. LaSCO cannot state system liveness properties such as obligations. Also, LaSCO cannot state policies relating to objects that are in more than one distinct state.

PolicyMaker [BFL96] and its successor, **KeyNote** [Bla2001], are verifiers which give yes/no answers to questions of the form 'is the proposed operation safe to execute in terms of the defined security policy?' (written as `key1, key2, key3 REQUEST actionstring`). The benefit of PolicyMaker/KeyNote is that policies (specifically, security policies) are defined outside of the application code, rather than hard-coded into applications, and can therefore be altered by responsible users [Bla2001].

The **Authorisation Specification Language (ASL)** [JSS97] is a logical language that allows the specification of access control policies, but does not prescribe an implementation.

The **Security Policy Language (SPL)** [RZF2001], used for database transaction control, is designed to express policies about the acceptability of events.

Access control frameworks, like integrity constraints (page 25), are merely validity checks, rather than action initiators: they support *prevention*, but not *intervention* strategies.

Network and System Management Policies

In the field of network and system management, policy-based systems incorporate event monitoring and policy triggering mechanisms. In all case, policies are laid atop a traditional, statically-typed object-oriented framework.

Bell Lab's **Policy Description Language (PDL)** [CL2001, CLN2000, LBN99] expresses event-condition-action rules with action concurrency constraints, but not access control policies. Extensions to Bell's PDL [CL2001] argue for history-based policies where epochs of events are maintained. Formal mathematical argumentation is provided but there is no discussion of the implementation intricacies of storing and monitoring persistent occurrences in database environments.

Koch's **Policy Definition Language (PDL)** [Koc97] relies upon the primitive EDL language (page 18) which can monitor only number buffers, but not abstract business events.

The IETF's **Policy Core Information Model (PCIM)** [MESW2001] provides a UML class structure for the representation of policy for the management of network devices. Policies are condition-action rules, where the action is typically a primitive of SNMP, CMIP, LDAP or other low-level network or system administration protocols.

Imperial College's **Ponder** [DDLS2001, DLSD2001] – perhaps the most representative of the policy management frameworks and hence the one we review in most depth here – can express authorization and 'obligation' policies as well as some basic descriptive policies through an associated domain service. Ponder is intended to improve system flexibility by not hard-coding policies into components. Sloman [Slo94, Slo95] argues for the need to specify, represent, and manipulate policy information independent from management components to enable dynamic change of policies and reuse of components with different policies. A policy service provides the means for storing policy specifications, determining the objects to which they apply, performing analysis to detect conflicts, and disseminating policies to managers for interpretation [Slo95]. Positive and negative authorization policies allow access control specification. 'Obligation' policies support event-triggered actions. Path-based domain scope expressions specify the sets of objects to which a policy applies [DDLS2001].

Critique

The implementation-level languages employed for access control and network management are ill-suited to specification or subsequent interpretation by business users. Policy-based frameworks are primarily intended for systems administration tasks, and are not appropriate for business workflow application development for the following major reasons:

2.3.1 Event storage services and retrospective review are not inbuilt

Many access control logics [JSS97, CL2001] and language implementations [DDLS2001, HV2001] support dynamic constraints such as *separation of duties* and *conflict-of-interest (Chinese Wall)*. Examples are:

(1) A person may not approve a cheque they have issued. [Dam2002, HV2001] or

(2) An author of a paper cannot act as its referee. [CDMR2001]

These policies, which are common in workflow scenarios, require that *past events, states, and execution history* be captured [BFA99, HPL99, JSS97, RZF2001]. While history-based policies are relatively easy to express, they are difficult to enforce efficiently due to the need to maintain and query a record of past events [RZF2001].

A stream semantics, where events are short-lived and rapidly consumed, is assumed by the event monitors in policy management frameworks. In contrast, e-commerce applications require persistent event histories. Notably, a generic event- and state-history storage service is absent from all access control logics and language implementations reviewed here, and developers wishing to implement history-based policies must find a means to maintain and monitor a persistent occurrence record themselves. In ASL, history-based authorisations, based on accesses previously executed by a requestor subject, can be specified, though no event storage or monitoring mechanism is described. Ponder requires, but does not provide, an event-history monitoring system in order to enforce history-based policies [Dam2002, p106]. In both Ponder and **Tower** [HV2001], an object-oriented information model is assumed. The cheque approval policy (separation of duty constraint (1) above) is implemented by modelling both the `issuerID` and `approverID` as attributes of the `Cheque` class. Generic information models and persistent storage services for events are not provided.

2.3.2 Static typing

The policy management frameworks surveyed here are all built atop object-oriented languages. In object-oriented environments, entity types are determined *a priori* by the analyst who designs a static class hierarchy. In commerce, typing is both *a priori* and *a posteriori*. An entity's type can be determined from the occurrences in which it has participated. To support a posteriori typing, information about event and state history must be available and queryable in an occurrence store. An item's type is dynamically determined by ascertaining whether it fits certain criteria, defined before or after the existence of the item.

Requirement 17	**It must be possible to define and store the criteria that an item must satisfy – the description (query) it must fit – in order for it to count as being of a certain type.**

Object-oriented development environments do not incorporate services to store event or state information nor are services for storing queries (sets of criteria) provided. Keeping a persistent log of occurrence information typically requires the insertion of OQL or SQL code in various methods of interest by the application programmer to keep track of events and states by explicitly writing them to persistent store.

Damianou [Dam2002; p40, p42] criticizes approaches such as LaSCO [HPL99] which only allow policies to be specified for classes in a static object-oriented hierarchy. Damianou remarks that, in practice, policies are specified for sets of objects grouped together for reasons other than the (static, object-oriented) class or type of the objects. ASL [JSS97] similarly provides no means for specifying authorization rules for groups of objects that are not related by static type hierarchy [DDLS2001].

Requirement 18	**We must be able to express policies applying to intensionally described (rather than merely extensionally listed) groups of objects. The members of such groups may change dynamically.**

In Ponder, the scope of policies is specified using domain scope expressions (descriptions), rather than merely pointers to class identifiers. Domains improve *flexibility* and *maintainability* by grouping objects for convenience and by shielding policies from changes in group membership. They offer *scalability* by allowing single policies to be uniformly stated for millions of objects in large systems, rather than requiring separate policies to be stated for each individual object, which would introduce consistency problems during policy update [Dam2002].

Ponder's deployment model [MSY95, DLSD2001] resolves domain scope expressions in order to determine all objects to which a policy applies. Policies are then pushed to all applicable objects. The Ponder service does not provide 'pull' facilities to allow objects to determine which domain scope expressions (and therefore policies) cover them.

Requirement 19	**Policy environments require a coverage detection service and interface that would allow objects to determine which of a changing set of descriptions they, or other objects, fall under.**

2.3.3 Analysis phase of system development is under-supported

A variety of authors have suggested that policies exist at various levels.

Masullo and Calo proceed down from societal policies, to organizational policies (including contractual agreements and quality programmes), functional policies, and process and procedural policies [MC92]. Goh [Goh97] describes a vague progression

from high-level objectives to low-level implementables. Michael, Ong, and Rowe [MOR2001] see a distinction between meta-policy (policy about policy), goal-oriented policy (which specifies an outcome but no action), and operational policy (which defines actions but not goals). Wies's levels [Wies95] descend from corporate (high-level) policy, to task-oriented, functional, and then low-level policies. Wies provides a policy template to store state and free-text information about policies, such as their author, creation date, goal or activity, status, life-time, and functional area. Koch [Koc97] mentions that there is a transformation from:

- natural language requirements-level policies, stored as text glosses with no semantic structure, to
- semi-structured goal-level policies, which are given a name, ancestor, descendant, subject, target, modality, action, constraint, event, no-success (exception / failure) clause, and status, and finally to
- formal, executable, operational-level policies.

Moffett and Sloman [MS94] do not advocate specific policy levels, but suggest there is a fuzzy progression from human-interpretable high-level policies written in natural language, to machine-readable low-level imperatival and authority policies. They illustrate how goals may be refined into sub-goals and how the relationship between sub-goal completion and goal completion can be represented in Prolog. Finally, Damianou [Dam2002] accepts three levels of policy specification for network and system management: high-level abstract policies (goals, service level agreements, and natural language statements), specification-level policies (network-level or business-level policies written in policy specification languages, rule-based specifications, and formal logic), and low-level environment and operating-system specific policies or configurations. Refinement from high-level policies into specification- or low-level policies is outside the scope of Damianou's work. Instead, he provides mechanisms for generating low-level LDAP entries, Java classes and policy files, and Linux and Windows access control specifications, from specification-level authorization and

obligation policies (written in the Ponder policy language). The generated output is deployed to network components.

Access control policies and system and network management policies are specified by system administrators. In contrast, policies in e-commerce are typically specified by business analysts. Policy management architectures are targeted at low-level network and system management applications as opposed to representing and enforcing e-commerce contracts and business policies. The transition from policies stated in natural language to statements that can be enforced by a machine is difficult.

Translation of a limited class of policy statements – access control expressions – written in Controlled English [Pul96] into OASIS Role Definition Language has been prototyped [Llo2000, BLM2001]. Michael, Ong, and Rowe [MOR2001] emphasize that being able to quickly translate a natural language specification of security policy into a formal logic would be useful as policy bases can be large, policy changes frequently, and the relationships between policies may be complex. Ideally, they argue, policy would be stored in a computational form in a centralized, searchable, and updateable repository. The authors describe their natural-language-input-processing tool: this is merely a part-of-speech tagger which maps natural language sentences expressing security policies to an object-oriented schema and allows queries against the schema. No mechanised interpretation and enforcement of the output is described.

Requirement 20	**Though it is perhaps not fully machine-automatable, the progression from English specifications to machine interpretable policies should be further systematized.**

2.3.4 Absence of a commercial notion of obligation

The 'obligation' policies mentioned in policy management environments use a non-commercial notion of obligation: *impersonal (collective) obligations* are not supported, *ought-to-be* policies are not expressible or checkable, components are not *asynchronously*

invoked or monitored as they are assumed to lack freedom of choice, *individual obligations* are not identifiable, and obligations are not distinguished from *powers*.

Personal obligations only

In high-level business policy, impersonal, *collective obligations* are common, as in:

> Customers must be notified (by some employee) of outstanding balances.

Here, no specific actor is specified but it is clear that the obligation can be fulfilled through appropriate notifications to customers. In Ponder, the use of 'some X' as the subject of an obligation policy is not supported as the deployment mechanism is push-based and it is unclear which object of type X to push a 'some X' policy to. In Ponder's low-level policy specifications, each policy expresses an activity invocation (personal obligation) on an object. It is important that impersonal obligations be formally expressible since companies frequently rely upon the initiative of independent agents to meet these requirements.

Requirement 21	**Impersonal (collective) obligations must be expressible and checkable.**

Low-level ought-to-do obligations only

Ponder is targeted at specifying low-level policies only and is unable to represent high-level goals: actions (method invocations) can be obliged, but not states of affairs. As Cole et al. point out [CDMR2001], this is unsuitable for instances where the states of affairs to be obtained or avoided are known, but the actions which would bring about such states of affairs are unclear. Deontic logicians have often emphasised the need to support both *ought-to-do* (in German, '*tun sollen*') and *ought-to-be* ('*sein sollen*') obligations [Kro97]. High-level policy specification languages must therefore be able to express obligations which speak of either actions or states-of-affairs.

Requirement 22	**It must be possible to express and monitor both obliged *actions* and obliged *states of affairs*.**

Synchronous invocation (no freedom of choice) only

Obligation policies implement traditional, synchronous rules which model real world obligations poorly (page 33). Critically, the assumption that subjects are well behaved and lack freedom of choice has led policy management architectures to neglect *fulfilment* monitoring. So-called 'obligation monitoring' in Ponder entails a form of obligation *applicability* monitoring, rather than obligation *fulfilment* monitoring; procedural scripts are invoked when applicable policies fire, with Ponder throwing exceptions upon failure and Koch's PDL invoking a `nosuccess` clause. Lee uses an `or-else` connective [Lee80, p142] to model reneging (defaulting). These exception-handling approaches make it onerous to subsequently and dynamically add (possibly conflicting) clauses which define what circumstances bring about fulfilment and violation, as is common in commercial legal reasoning. For instance, the introduction of the *de minimis* rule in English law provides that a marginal discrepancy in performance against that contracted does not amount to violation of the corresponding obligation [TB99, p144]. Approaches based on exception throwing, invocation of a `nosuccess` clause, or introduction of an `or-else` connective, impose a substantial maintenance burden when broad-reaching legal provisions redefine what is meant by 'fulfilment' and 'violation'. Procedural enforcement approaches force an arduous search-and-inject process where applicable exception code would need to be located and modified in all relevant places.

Requirement 23	**The implementation must allow introduction of broad reaching provisions (e.g. defining 'fulfilment' and 'violation') by a single insertion anywhere in the specification.**

General obligation policies only (no identified obligation instances)

Perhaps the most problematic aspect of Ponder's representation of obligations, however, is the granularity of treatment: Ponder identifies *general* obligation policies, but not *individual* obligations. For example, the obligation (policy) of nurses to monitor, is not separable into the individual obligation of Sister Mary to monitor Dirk, or the obligation of Sister Agnes to monitor Rachel. As no handle is available to these individual obligations, the case-specific obligation instances of general obligation policies cannot be voided without voiding the entire set of obligations. Ponder supplies only a course-grained `disable()` method for general obligation policies, not specific obligation instances, and also maintains no audit trail of which policies were disabled or fulfilled and under what circumstances.

Requirement 24	**We must be able to refer to individual obligations of each party, and trace each obligation to the general prescriptions, and events, that brought it about, or that terminated it.**

No distinction between obligation and power

The SPL security policy language [RZF2001] claims to implement a restricted type of 'obligation', expressed informally using sentences of the form:

(To make Action_T successful)[4]
Principal_O *must do* Action_O
if Principal_T *has done* Action_T

or

Principal_T *cannot* (successfully) *do* Action_T
if Principal_O *will not do* Action_O

Enforcement is through an access control service and security monitor. The security monitor allows a `commit` on a database transaction containing event `Action_T` only if the execution history contains `Action_O`. SPL's notion of 'obligation' is therefore more accurately a form of *history-based permission* or *pre-condition*. SPL 'obligations' are constraints enforced by security monitors, and not obligations for agents to execute specific actions on the occurrence of events [Dam2002, p38]. Though [RZF2001]

[4] Clauses in brackets have been added to clarify the intention of [RZF2001].

does not recognize this, SPL policies are more closely related to the concept of legal power [JS96] than to the concept of legal obligation. However, their approach to implementing 'powers' does not provide that a persistent, queryable record of attempted (not-yet-committed or already-rolled-back) action be kept; retrospectively determining which actions did not 'commit' is therefore not possible as this information is discarded. Further, it fails to distinguish between the originating event (e.g. a physical return) and the consequential legal event (e.g. a return in terms of Clause C.3) that it brings about. This is troublesome as it makes it impossible to express conflicting opinions, such as the case where one clause sees the existence of a legal return originating from the physical return, whereas another denies the existence of a valid return. Furthermore, failure to distinguish between the originating physical events and consequential legal events means that denying the existence of the legal event necessarily denies the existence of the physical event. This has the undesirable effect of obliterating actual physical history. Consider that, in a legal scenario, attempted 'returns' should not be omitted from the record merely because they never qualified as 'returns in terms of Clause C.3'. This omission happens in SPL's implementation device.

Requirement 25	**Physical occurrences must be distinguished from legal occurrences. Both must be recorded.**

2.3.5 Configurable conflict resolution is not provided

In Ponder, meta-policies control access to other policies in the database, thereby providing some measure of conflict *avoidance*. However, the conflict *resolution* facilities supported are limited. A fixed, domain-nesting-based precedence is used to resolve conflicts, effectively meaning that specificity-based precedence is used. Negative authorization policy (prohibitions) are taken to always override positive policies (permissions and obligations) [MSY95]. This is a limiting assumption given that in many cases the reverse may be true: in many practical scenarios, a prohibition may be violated in order to fulfil an obligation. In commerce, the decision as to

which policy prevails is context-dependent and must be made dynamically, based on criteria such as recency, specificity, location, authority, leniency, or cost/benefit evaluation. Since Ponder captures information such as policy author in opaque C-language-style comments [MSY95], rather than in interrogable attributes, conflict resolution cannot be automatically performed since important information about the policy is not semantically accessible.

In ASL [JSS97], integrity rules may be used for conflict *avoidance*, to prevent contradictory access control policies from being specified, by raising an error. Resolution rules specify whether, in the case of conflicts, permissions or denials take precedence for a given set of subjects and objects and a given type of access operation. Conflict *resolution* is therefore more fine-grained than applying a single system-wide precedence policy as is used in Ponder [DDLS2001]. As in many other approaches [HPL99, UM96], rules in ASL are not labelled, so cannot be referred to by identifier or name. Because ASL assumes that all rules are stated by the System Security Officer, the identities of the utterers of rules are not recorded. The model needs to be extended to reason about conflicts between rules stated by different groups of system administrators, or to reason about conflicts between user-defined rules and supervisor-defined rules. Such conflicts are common in distributed settings where management is semi-autonomous.

Many frameworks [UM96, RZF2001] do not provide any control over the policy-life cycle, as creation of a policy necessarily involves their activation, and de-activation (voidance) cannot be achieved without deletion. Conflict resolution through control of the policy life cycle is therefore not feasible. As in many general rule-based approaches, priority of access control rules in **SLAPD** [UM96] is hard-coded in the ordering of the rules: conflict resolution is achieved by applying the first matching rule and ignoring remaining rules. Rule selection policies for advanced configurable conflict resolution cannot be explicitly specified.

In the IETF's Policy Core Information Model, priority is indicated with a non-negative integer. It is assumed that policy is centrally and consistently specified, and

the policy originator cannot be recorded, so multiple clients cannot state different policies for a device.

2.4 Business Process Modelling & Animation

Animation of business process models during e-commerce application development is the task of *workflow management systems*. With *executable specification* technologies, the intention is to yield a declarative, machine-interpretable business logic.

Workflow Management Systems (WFMSs)

Workflow management systems (WFMSs) are targeted at process modelling. Constructs generally follow those provided by the **Process Specification Language (PSL)** [SGTV2000] and the **Workflow Management Coalition's (WfMC) Workflow Process Definition Language (WPDL)** and reference model [WfMC95]. Workflow specification is based on process synchronization, with activities linked via transitions using `and/or`, `-split`, `-join`, and `iteration` primitives [GR2000, vdA96]. The WfMC reference model is adopted by the OMG's **Workflow Management Facility** [Sch99]. In **ADEPT** [JFN2000], the business process definition language uses comparable primitives: `sequence`, `can-be-parallel`, `must-be-parallel`, and `loop (iteration)`. **TriGSFlow** [KRR98] provides sequencing, branching, and joining primitives using the logical operators `AND`, `INCLUSIVE-OR`, and `EXCLUSIVE-OR` to create composite event expressions.

The **Orchestra Process Support Service (OPSS)** [CDNF2001] uses the event composition operators `sequence`, `or`, `and`, and `not` drawn from the composite event detection language Snoop (page 15). The **REWORK** workflow system architecture incorporates the Eve distributed composite event detector (page 14). Composite events in Eve are specified using the operators `SEQUENCE` (event of type E1 followed by event of type E2), `CONCURRENCY` (events of both types occur at virtually the same time point), `REPETITION` (*n* occurrences of E1), `ITERATION` (all events of type E1 until an event of type E2), `DISJUNCTION` (first event of any of the mentioned types),

`EXCLUSIVE-DISJUNCTION` (one event of any of the mentioned types in interval), `CONJUNCTION` (events of both types), or `NEGATION` (no event of type E1 within an interval). Absolute, relative, and periodic time events can also be defined. Composite event expressions may be tagged with the atoms `:same-workflow` and `:same-broker` to require that the events occur in the same workflow or have the same broker as their origin, but general parameter bindings between events in a composite expression are not supported so it would not be possible, for instance, to require that all matched events pertain to the same user or object.

Several workflow management systems, such as **SAP R/3**, **ARIS**, and **BaanERP**, base their process modelling languages on Petri Nets [GR2000, SvdAA99]. **Petri Nets** are event-based formal graphs for modelling concurrent processes and systems [Sow2000]. Tokens, which carry with them associated data, are passed around the net. Tokens rest at 'places' which represent states. Transitions, which are connected to places via directed arcs, fire when all their input places have a token and guard conditions are met, thereby implementing synchronization. Firing causes tokens to be removed from input places and relocated to output places. van der Aalst et al. [vdAvHH94] propose that timed, coloured, hierarchical Petri Nets may be employed for the executable specification of sequences of states in workflows. Each token represents a job or resource, and the current place of the token captures the current state of the job or resource. Time is associated with transitions and each token has a timestamp that models the time the token becomes available for consumption. Each job has an identifier, a description, and a due date; hence, the tokens are called 'typed' or 'coloured'. Subsystems are modelled through hierarchical nesting of Petri Nets within others.

Executable Specification

Moving to higher-level and more natural mechanisms for specifying e-commerce systems is a desirable end-goal. Specifications should preferably be executable [Fuc92]. **Attempto Controlled English (ACE)** [FSS98, FSS99] is a controlled natural language (a subset of English) suitable for specification. ACE is computer processable and can be translated into a logic specification language which can be input into a theorem prover, and can be queried. Ambiguity is resolved through the useful mechanism of paraphrasing and user-confirmation.

Critique

Workflow and executable specification techniques for business process animation suffer the following drawbacks:

2.4.1 Process models show task dependencies, not legal relations

In conventional workflow implementations, the notion of business contracts is overlooked. Workflow systems encode a process specification, and concentrate on specifying task dependencies – ordering constraints – rather than interpreting a contract. No attention is paid to legal constraints such as rights and powers of parties. As a queryable occurrence history is not maintained, it is not possible to assess the legal implications of actions in the light of historical circumstances.

The ACE specification language does not allow the use of modal operators such as 'can', 'must', or 'may', which are typically used for the specification of permissions and obligations. The absence of notions such as creation, satisfaction, and violation of commitments means that the logical form employed by ACE does not model the legal primitives prevalent in electronic commerce specifications.

Requirement 26	**Provisions (legal relations), should be explicitly stored, not implicitly encoded in process models.**

Lee et al. [LD92, Lee98, BLWW95] propose to execute trade procedures by extending Petri Nets with deontic (legal) states of affairs brought about at each place. Their approach is targeted at modelling document and communication flows and their legal effects, rather than executing contracts. Furthermore, the approach continues to suffer from the drawback that Petri Nets are stateless by default, maintaining no queryable occurrence history; this makes it impossible to assess legal implications of past actions retrospectively.

2.4.2 Rigid communication and obligation creation protocols

Negotiation protocols set up agreements following a sequence of capability advertisements, offers (proposals), counter-offers, rejections, and acceptances. They are regularly discussed in the agent communication languages used in the business process management literature – e.g. in the ADEPT framework (page 47), and in [FK2001]. ADEPT provides that a message '`I propose …`' necessarily brings about a valid offer, and '`I accept …`' necessarily brings about a commitment, irrespective of circumstances and background law.

In contrast, Jones's contention [Jon2002], also embraced by Kimbrough and Tan [KT2000], is that the force of an utterance is not innate, but rather derived from *social conventions and background law applied by the interpreter*. 'I propose …' and 'I commit …' then bring about an offer or commitment (prima facie, according to a particular law) not merely when the utterance is made alone, but also when the 'fidelity conditions' [Aus76, SV85, Cru2000, Tho98] mentioned by the laws that define valid offers and commitments are satisfied. The laws to be applied to determine the illocutionary force are chosen by the *interpreter*, so the force is not hard-wired by the speaker alone as is usually assumed in traditional agent communication languages and protocols.

Requirement 27	**Existence of obligations must be derived from interpretation of law, rather than from a closed set of communicative acts or a rigidly defined protocol.**

ADEPT's negotiation protocol relies on the assumption that an agent is never obliged to accept a request from a peer. While this assumption is consistent with a theory of free-willed agents and is often useful, it hard-wires the governing legal framework, and ignores the possibility of option agreements, umbrella contracts, standard terms and conditions, or relationship governance letters. These are common in industry, and are used for specifying the conditions upon which future contracts are formed: they are therefore sometimes termed 'governing contracts' in legal parlance. Governing contracts simultaneously confer powers to parties and impose legal liabilities upon counter-parties. For instance, an option or conditional contract [TB99, p12] may give a party the power to buy a certain amount of steel at a future date. In this instance, the option holder can exercise their option (legal power) and create an obligation on the supplier merely by making a request – there is no requirement that the request be accepted in order to create a valid contract. The obligation to deliver the steel comes into effect upon the occurrence of the event of requesting, irrespective of whether the option-bound supplier accepts or not. In ADEPT, the architecture has ingrained within it the assumptions of a single rigid system of law.

Requirement 28	**The model should provide fundamental legal conceptions, rather than hard-code the constraints of a particular system of law.**

2.4.3 State history is not accessible

van der Aalst [vdA96] argues that a state-based approach complements event-based techniques as we need to refer to states such as 'enabled' and 'executing'. States make it easier to handle job transfers and rerouting as tokens can simply be moved to the relevant place on a compatible procedure in another WFMS. Other authors [CDDF98, CDNF2001] agree that event-based information must be supplemented with state information, since event-based synchronization works only when the

components that need to synchronize are active and ready to receive synchronization events.

OPSS (page 47) makes use of a State Server. The State Server provides information on the current state of process entities; it maintains the current state by subscribing to events of interest. Historic state is not stored, and only a predefined list of processing entity states is provided: these include `Available`, `NotAvailable`, `Defined`, `Assigning`, `OnGoing`, `Suspended`, `Terminated`, and `Aborted`.

ADEPT (page 47) employs the trivalent condition values `true`, `false`, and `unknown` to denote that a task has been successfully completed, has failed, or either has not started or is in the process of being executed. Non-commencement and in-progress states are indistinguishable in ADEPT as they are both indicated using `unknown`. Only the current state is accessible, and past states are not recorded.

Attempto Controlled English (page 49) addresses specification-time natural language understanding technologies and provides no persistence services. It provides neither implementation-level services for event and state recording and monitoring, nor persistent storage of specifications or run-time data.

2.5 Implementations of 'Contracts'

Much recent work in procedural environments claims to employ the notion of a 'contract'.

Critique

The term 'contract' is typically used metaphorically; the conditional obligations, permissions, and powers that define legal relations between parties and characterize business contracts [Hoh78] are seldom captured. Following are some of the restricted perspectives of the notion of a 'contract' employed in the literature:

2.5.1 Object-oriented constraints perspective

In the object-oriented programming community, Meyer [Mey97, Mey99] has advocated **Design by Contract**. By Meyer's own admission [Mey97, p127] the word contract is used analogically. Meyer's 'contracts', implemented in Eiffel, define the 'rights' and 'obligations' of method routines in classes through pre-conditions (`require` clauses) and post-conditions (`ensure` clauses). Class invariants (`invariant` clauses) specify object consistency properties. The intention of each of these Boolean assertions is that a 'supplier' class be able to guarantee to its 'customers': "if you promise to call a method routine with its pre-condition satisfied, then I, in return, promise to deliver a final state in which the method post-condition and class invariants are satisfied".

Beugnard and co-authors [BJPW99] provide a general taxonomy of the types of 'contracts' employed in object-oriented software development:

- Basic Contracts: are non-negotiable interfaces defined in an interface definition language such as CORBA IDL. These contracts specify operations a component can perform, as well as input and output parameters, and possible exceptions raised.
- Behavioural Contracts: are non-negotiable method pre-conditions, post-conditions, and class invariants (Boolean assertions) as available in OCL, Eiffel, iContract, and KAOS (page 25). Though not mentioned in [BJPW99], Holland [Hol92] combines interface definitions (Basic contracts) for multiple classes with multi-class invariants (Behavioural contracts), to form 'contracts' used as a module interconnection language. Holland's contracts have the form:

```
contract [contractName]
        [className] supports
                {attribute signatures}
                {method signatures}
        …
        invariants
                …
```

```
end contract
```

Holland's goal is to ensure reusability of relationships and interconnections between classes by ensuring individual classes comply with their contracts (interface definition and invariants) as the system evolves.

- Synchronization Contracts: are non-negotiable synchronization policies, expressed through `mutexes` (semaphores) and Java's `synchronized` keyword which prevents concurrent invocation of a block or method.
- Quality of Service (QoS) Contracts: are negotiable parameters such as maximum response delay on method invocation, average response, result precision, or data stream throughput rates. These are implemented through QoS extensions to component frameworks such as CORBA.

Beugnard et al. define contract management as incorporating contract definition, subscription, application, termination, and deletion. Application involves runtime monitoring. Contract violations are handled through raising an exception, waiting or retrying until a precondition is satisfied, or simply ignoring the violation. For Basic, Behavioural, and Synchronization contracts, definition and subscription are indistinguishable as they occur at the build-time of the component. Similarly, termination and deletion are inseparable and occur when components are unloaded. In QoS contracts, defined contracts can be tailored to dynamic operating conditions as clients can select a conformant service that has advertised properties within the required threshold, and can 'subscribe' by setting service parameters. It is suggested that acquiring the ability to alter contracts in response to changing needs is strong motivation for defining QoS contracts as objects. Beugnard et al's discussion illustrates the many manifestations of contracts in object-oriented platforms. The mapping of these diverse technical implementation constructs to business contracts is obscure, and translation from the former to the latter consequently unsystematic. 'Contracts' are haphazardly distributed across various implementation devices, with no consistent mechanism of finding, checking, and altering them.

2.5.2 Task allocation or process co-ordination perspective

In Smith's **ContractNet** protocol [Smi80], high-level task descriptions are advertised (announced) to distributed nodes in a processing 'net', contractors bid for subtasks, and managers award contracts to winning bidders. The protocol, which Smith says 'resembles contract negotiation', is used for task allocation. Weigand and Xu [WX2001] comment that the protocol is useful in the context of distributed problem solving, but too naïve for situations where agents are involved in real business. The protocol is intended for problems that lend themselves to decomposition into a set of relatively independent subtasks with little need for synchronization. ContractNet is therefore inappropriate for business workflows, where task dependencies are common.

Contractual Agent Societies [Del2000, DK99] is an approach that extends the ContractNet protocol with electronic social institutions such as commitment monitors, notaries, reputation servers, and socialization services. These are expected to promote robust, stable, and efficient systemic behaviour. The goal is to evaluate the effect of these social institutions on average task completion times in a ContractNet-style allocation of tasks.

Minsky and Ungureanu propose a concept of **Law-Governed Interaction (LGI)** for agent communities [MU2000]. Agents that need to be co-ordinated with other agents choose to join a 'law' held by a controller component. The law mandates the effects of events such as message sending, arrival, and exceptions, but is not sensitive to the internal behaviour of agents, or to changes in their internal state. The law is intended solely as a shared record of the *coordination protocol* between agents, with 'obligations' implemented as named timers which cause the controller to revoke resources when a given number of seconds expires, unless `repeal()` has been invoked on the timer in the interim.

2.5.3 Service advertisement and invocation perspective

Contract-driven inter-enterprise workflow architectures such as **COSMOS** [MGTM98, Mer98], **CrossFlow** [KGV99], and **E-ADOME** [KCK2001] focus on service advertisement and invocation for pre-existing workflows, but attend neither to the problems of assessing legal consequences of actions against contracts and legislation, nor to ascertaining consistency between contractual terms and business policies. The COSMOS workflow engine invokes functions in accordance with temporal constraints extracted from contracts and assumes conflict-free specifications, failing to recognize that clauses may mandate conflicting occurrences and choice of applicable clauses in the circumstance may be necessary. COSMOS has been criticized for ignoring the possibility of deviation from expected behaviour and inability to reason about legal powers or consequences of violation, such as secondary obligations coming into force [DDM2001]. CrossFlow and E-ADOME use contracts for inter-organizational workflow process integration. Contracts describe the agreed workflow interfaces as activities and transitions, based on the WfMC's WPDL (page 47). Contracts also specify what data objects in the remote workflow are readable or updateable.

Milosevic et al's **Business Contract Architecture (BCA)** [MBBR95, Mil95] assumes that contracts are provided a priori; introduction of terms subsequent to initial contract input is not supported and the architecture is therefore unable to cater for dynamically changing business and regulatory environments. Contract validation is a once-off process at contract-input time, relying upon hard-coded checks embedded in the contract validator component. For instance, the BCA uses compiled locale-specific code to check for the existence of consideration in the contract; this makes the BCA inappropriate for systems of law that do not insist upon the exchange of consideration for the valid formation of contracts. For instance, in British law, deeds do not require the exchange of consideration[5]. In the BCA, invalidation of the contract or change to validation rules subsequent to initial

[5] Deeds are contracts that are explicitly expressed to be deeds, and are executed in a prescribed way: signed, sealed, and delivered [TB99; p7, p51].

contract formation is not possible. The BCA does not provide generic occurrence monitoring facilities, expecting each application developer to develop bespoke monitoring code to detect and signal non-performance to the contract monitor. Contract enforcement in the BCA is limited to either (a) signalling violation or (b) preventing the non-performing party from entering further contracts; the former is too weak and the latter too heavy-handed and the approaches are therefore rarely used in practice. The common business practice, which is not supported by the BCA, is to ascertain what legal relation between parties comes about in terms of the contract as a result of the violation.

SeCo [GSSS2000] stores contracts in signed XML containers. It bases its architecture on the BCA, and suffers similar drawbacks. SeCo's monitoring service monitors and logs only negotiation-phase events, not settlement-phase events. Negotiation-phase events are logged to the XML container, which stores the parties, product descriptions, and payment and delivery conditions for the contract. SeCo does not implement an enforcement service, nor interoperation with payment and logistic services. In order to attain those goals SeCo would need to encode product description and payment delivery condition semantics more formally.

Hewlett Packard Laboratories [MSM2001] present work-in-progress towards a high-level architecture for regulating electronic marketplaces using contracts, proposing a contract repository, validator, monitor, and evidence store, which is similar in structure to the BCA. The conceptual model remains to be implemented, with the authors planning to embody contracts in XML.

Milosevic's BCA and derivative approaches are inspired by the International Standards Organization's **Reference Model on Open Distributed Processing (RM-ODP)** [ISO95] which establishes requirements for new specification techniques. The RM-ODP emphasises the importance of contracts, obligations, permissions, and prohibitions, but, like the BCA, does not provide guidance as to the explicit representation of these entities. The RM-ODP does not incorporate concepts of business contract validation, monitoring, and enforcement [Mil95, p177]. The temporal nature of obligations is also not treated [CDMR2001]: policies that are

conditional upon certain events are not dealt with, though absolute time intervals can be used to specify policy applicability. The RM-ODP deliberately provides a cursory, high-level view of concepts related to contracts.

To fill the need for an appropriate, concrete notation for more rigorous specification of policies in ODP enterprise specifications, Steen and Derrick [SD2000] construct a UML model and a specification language for the RM-ODP Enterprise Viewpoint concepts. Their policy language is a combination of structured English and simple predicate logic, which they translate into the formal object-oriented specification language, Object-Z. Though the authors argue that the purpose of policy is to both 'constrain the *behaviour* and *membership* of communities' [SD2000, emphasis added], they limit their focus to behavioural constraints only. Membership constraints are not dealt with: policy statements in their notation are glued directly to static object-oriented classes, rather than to described sets of objects. A procedural, object-oriented mapping for behavioural constraints is used: permissions, prohibitions, and obligations map to pre-conditions on method invocations. Only a partial implementation of obligations is provided, as the obligations do not explicitly specify the events upon which the actions must be executed [Dam2002, p46]. Policies are labelled for discussion purposes, but neither policies nor the obligation instances derived from them have identifiers in the eventual formal specification. There is no way to obtain a handle to a particular obligation of a particular object, nor to determine its provenance and originating occurrence. It is not possible to define what is meant by 'violation' as would be required in different systems of law. For example, returning a book 1 minute after the deadline probably does not count as violation according to the *de minimis* rule of English law [TB99, p144], but may count as violation under other systems or provisions of law. It is not described how particular obligations may be voided, say, in the exceptional case where the Chief Librarian waives the obligations of borrowers whose books were destroyed in an accidental residential fire. Powers [JS96] and immunities [Hoh78] are not dealt with. The semantics for permission does not capture legal subtleties from the jurisprudential literature, such as the distinction between vested liberties – where others are forbidden from interfering

with the permitted action [Lin77] – and privileges – where indulging in the permitted action does not bring about violations [And58]. Detection and resolution of conflicts is not explored in any depth [SD2000, CDMR2001]. In terms of conflict resolution, permissions are viewed as combining conjunctively. Temporal override is hard-wired: it is assumed that subsequent permissions restrict the scope of earlier permissions. This conjunctive combination of permissions is a consequence of encoding permissions (and prohibitions) as pre-conditions of methods, and using object-oriented inheritance to cascade pre-conditions through sub-classes. The ultimate aim of the work is to develop a prototype tool for the specification and analysis (rather than execution and monitoring) of open distributed systems according to the ODP viewpoint. The authors speculate that once the specification can be stored electronically, model-checking techniques could be employed to verify if some actual enterprise conforms to a policy specification. Mapping high-level policies to implementations is left for future work.

The XML-based **Trading Partner Agreement Markup Language (tpaML)** from IBM Research [DDKL2001] uses 'trading partner agreement' as synonymous with 'contract'. tpaML is now pursued under the **OASIS Collaboration Protocol Profile (CPP) and Agreement (CPA) specifications** [OAS2002]. tpaML and CPP/CPA capture the technology specific interoperation parameters agreed by parties. These include message formats (e.g. OBI), encryption techniques (e.g. SSL), and communication protocols (e.g. HTTP). There is no notion of the rights and duties of the parties, nor provision for fulfilment monitoring. Furthermore, the framework does not provide for the management of the potentially conflicting policy sources that may govern a single business entity.

The goal of the **OASIS Provisioning Technical Committee** [OAS2001] is to propose standards for service provisioning. Their notion of a 'provision' is in the sense of 'providing resources'; the intention is to facilitate resource allocation by setting up, amending, and revoking system access rights (cf. *Access Control Policies*, page 34) to electronic services. This can be contrasted to the normative, contractual sense of 'provision', which specifies desirable and undesirable situations in terms of

conventions for interpreting various happenings, and attitudes towards the conventionally described occurrences. By dividing the problem into specifying inter-operation between separate provisioning systems, and specifying inter-operation between a provisioning system and its managed resources, they do not focus on the introspection required within any given provision management system to manage conflict situations.

Service Level Agreements (SLAs) are a form of contract. Terms of service for an existing workflow application are advertised, client agents locate suitable services and negotiate service parameters within certain bounds, and SLAs encode the agreement. The ADEPT agent-based business process management architecture (page 47) uses the SLA approach. In ADEPT, a fixed, standard template for SLAs is provided, which can be populated with parameters for each agreed service. For each client, the service name, number of allowed service invocations, cost per invocation, permissible invocation interval, and penalty incurred by the server for any violations, can be set. Costs and penalties are specified simply as integers in undefined units (e.g. 30). Complex pricing and exchange schemes, and penalties graded according to the nature, severity, and circumstances of violation, cannot be specified.

In spite of the presence of similar legal conceptions across contracts, internal policies, and external regulations, the contract-driven service advertisement and provisioning architectures surveyed here encode standard, parameterizable contracts as data, but leave policies and regulations buried in procedural code or workflow scripts. This makes it impossible to check the consistency of inter-organizational provisions (contracts) against intra- and extra-organizational provisions (policies and laws).

Requirement 29	**Provisions, whether emanating from contracts, policies, or laws (inter-, intra-, or extra-organizational provisions), should be uniformly represented, to facilitate consistency checking.**

2.5.4 Project management perspective

Dowson [Dow87] presents a project support environment, **ISTAR**, a language-independent software development project support environment. In ISTAR, contracts are used for project management of the software development process. That is, contracts are used for co-ordination of the developers and development process, rather than for executable specification of the software system itself.

2.5.5 'If-then' rule perspective

Grosof and co-authors [GLC99, RGW2002] specify contracts using declarative `if-then` conditionals but do not explicitly specify the legal relations of parties. Their **ContractBot** system employs parameterizable agreement templates ('proto-contracts') and constraint rules to automate auction-based negotiation. Proto-contracts are populated with the auction results, which provide values for buyer, seller, price, quantity, and other attributes. The instantiated contracts can then be executed as SCLPs (page 26).

2.5.6 Financial-domain-specific perspective

Peyton Jones et al. [PES2000] use the functional language, Haskell, to represent a small class of contracts: financial option contracts where performance is by cash flows only. No representation of obligations is provided, and the representation is used only for option description and valuation, and not for contract execution.

In contrast, Lee's formal **CANDID** calculus concerns itself not with valuation of a small class of financial contracts, but with the formal description of a variety of such contracts [Lee80; p120, p184]. CANDID describes leases, call and put options, insurance, debt instruments (registered and coupon bonds, and secured loans), equity instruments (common and preferred stock), and convertibles. Lee envisions extending CANDID to the representation of more general contracts, such as construction contracts and product warranties [Lee80, p2]. Automated monitoring of contracts, completeness and consistency checking, and impact analysis of changes

are the desired eventual applications of the calculus, though the emphasis of CANDID was primarily on modelling rather than on computational aspects [Lee80; pv, p190]

2.5.7 Legal perspective needed

Huhns and Singh [HS98] remark that a system that supports Hohfeld's conceptions (op cit, page 31) could represent contracts and would be useful for defining and testing the compliance of agent interactions to norms. The discussion above demonstrates that many software implementations that claim to use 'contracts' use the term in a sense quite unlike its business connotation, failing to explicitly or generally capture rights, duties, and legal consequences of violations, and providing only limited, parameterizable agreement templates.

A notable exception is recent theoretical work by Daskalopulu, Lee and others, which assesses the status of legal contracts and the implications of events based on a Finite State Machine [DDM2001, DM2001] or Petri Net [BLWW95, Das99] approach. Establishing the procedure and deriving the Petri Net from the business contract is, however, non-trivial [Das99]. Such approaches can only be appropriate if the reduction to a particular FSM or Petri Net will remain valid for all time, which is unrealistic in the volatile world of business contracts. FSM and Petri Net-based techniques reduce contracts to directed graphs, which capture the business procedure, but leave provisions, and occurrence history, implicit.

Requirement 30	**An approach is needed where provisions are explicitly captured as data, and are thus readily available for inspection and analysis.**

Explicit storage of provisions would allow consistency checking, contract performance assessment, and management review of which provisions pertain to which items or occurrences.

2.6 Deontic Logic

'Deontic logic' has come to describe various subtly differing logics of norms, or more specifically, logics of obligations and permissions [MW93], and ideality versus actuality [Ser99]. These are mostly derived from initial proposals by von Wright [vW51]. The standard system of deontic logic augments propositional logic with the operators O, P, and F, for obliged, permitted, and forbidden. As a broadly representative axiomatization of deontic logic, we have chosen some principles from the system described in [MW93] as the focus of discussion here:

Interdefinability:	$P\alpha =_{def} \neg O \neg \alpha =_{def} \neg F \alpha$
'α is permitted' is equivalent to 'it is not obliged that not α' and 'α is not forbidden'.	
Strong consistency theorem:	$\neg(O\alpha \wedge O\neg\alpha)$
It is never the case that α is both obliged and forbidden.	
Deontic detachment / transitivity:	$O(\alpha \rightarrow \beta) \rightarrow (O\alpha \rightarrow O\beta)$
If it is obliged that (if α then β) then, if α is obliged then β is obliged.	
Ought implies can:	$O(\alpha) \rightarrow P(\alpha)$
If α is obliged then α is permitted.	

The possible world semantics for Standard Deontic Logic (SDL) says that something is obliged if and only if it is the case in all deontically perfect alternative worlds, and is permitted if and only if it is the case in at least one such world.

Critique

The ideal worlds semantics of SDL has been criticised on a number of bases, which we deal with in turn in the following subsections:

2.6.1 Ideal world semantics: The moral 'ought'

Statements in SDL speak of what is the case in an ideal world: the modal operator O is better yielded as the fanciful moralistic notion of 'in a perfect world, it ought to be that …' rather than as the commercial contractual concept of 'there is an obligation

to …'. In commerce, violations and external events happening in the often-imperfect real world force the imposition of particular new obligations and the termination of old ones. SDL's *deontic detachment principle* (page 63) and ideal world semantics results in some paradoxes, such as [Mak99]:

> "The clerk ought to give you a receipt if you pay" and
> "You ought to pay" implies
> "The clerk ought to give you a receipt" (even if you don't pay).

This paradox arises as SDL mandates what must be the case in an ideal world, but says nothing of what happens in an imperfect world, when things go wrong. SDL *fails to address contrary-to-duty obligations*, which are secondary obligations that arise from the violation of a primary obligation [PS97].

2.6.2 Conflict-free specifications

The *interdefinability* axiom and *strong consistency theorem* (page 63) imply that a state-of-affairs cannot be both obliged and prohibited at the same time, and that obligation to do something implies permission to do it. Sloman [Slo94] comments that, in reality, obligations and authorizations can be specified independently, although obligation without authorization can lead to conflicts. Taveter and Wagner [TW2001a, TW2001b] add that the reduction of permission to absence of prohibition is an unrealistic oversimplification. Chellas [Che80] argues that inconsistent norms and conflicting obligations ($O\alpha$ and $O\neg\alpha$) abound. SDLs axioms fail to account for the fact that obligations and permissions may arise from different sources, and may therefore conflict. A number of types of conflict need to be attended to:

Conflict with defeat (prima facie versus all-things-considered obligations)

Kimbrough and Moore [KM93] remark that genuine conflicts of obligation are not only possible, but common, and suggest the way out of this dilemma lies in recognizing that most obligations are defeasible. A consequence of adopting the strong consistency theorem, as well as the monotonic nature of SDL, is that conflict of norms and *prima facie* or *defeasible* obligations are unaccounted for [HK93, KM93,

PS97]. An example might be the obligation to give a 10% discount to loyal customers, which is defeated by the obligation to give a 15% discount to platinum customers (who may also be loyal, but should not qualify for both discounts). Kimbrough and Moore [KM93] propose a 'presumably' operator: $POα$ is taken to mean 'presumably $α$ is obliged'. Kimbrough and Moore's presumption operator, P, however, is undirected, making it impossible to specify who makes the presumption and on what grounds. This makes choice of a winning presumption difficult.

Conflict without defeat (necessary violation)

A complementary avenue for attending to conflict is to recognize that conflicting obligations may arise from different sources, and fulfilment of an obligation to one party may require violation of an obligation to another. Prohibition to do something does not necessarily imply permission to refrain from it as encoded in SDL, since Catch-22 situations exist where violations are unavoidable: "damned if you do, and damned if you don't"[6]. SDL does not address *conflict of duties*: for instance, for a bank customer with a low balance, the obligation to pay a debt may conflict with the obligation to stay out of overdraft, and this may require prioritization of obligations [dAMW96].

Conflicts arising from document provenance and occurrence evidence

Cholvy and Cuppens speak of merging sets of norms originating from conflicting roles [CC95] or regulations [CC98]. They extend SDL by relativizing norms to regulations, expressing sentences of the form '*regulation R says*, it is obligatory / permitted / forbidden that P'. Their approach is reasonably course-grained as all norms in regulation *R1* are taken to override all norms in regulation *R2*. Their logic therefore does not attend to conflicting obligations brought about by different events covered by the same norm. For example, Hansson and Makinson [HM97] give the case where the single rule that 'the doctor must immediately visit heart attack victims' generates conflicting obligations when two remote patients suffer heart-attacks: an obligation to visit patient *A* immediately and a separate, conflicting one to visit

[6] This phrase is attributed to Lorenzo Dow (1777-1834).

patient *B* immediately. Cholvy and Cuppens' treatment of the *source or origin of obligations* (and conflicts) effectively looks at their *provenance in documents* but not at their *birth in circumstance*.

2.6.3 Directedness of obligations and permissions

SDL abstracts away from agents [Wag2001] and concentrates on impersonal statements, rather than *personal* statements that are associated with particular action performers [CDMR2001]. This downplays the importance of policies as encoding relationships between explicit subjects and targets [Dam2002, p34]. SDL does not account for the directedness of obligations from the right-bearers to the counter-parties [HK93]. Various authors [HK93, TT99] have attempted to remedy this through the introduction of indexed deontic and action operators such as ${}_xO_yE_x\alpha$ meaning x is obligated, to y, that x bring about α.

SDL does not express who *issued* the obligation or permission. Such information is needed for conflict resolution. A consequence of its failure to assign a deontic opinion to the person expressing that attitude is that SDL requires that some normative stance (permission, obligation, prohibition) is taken with respect to every state-of-affairs α. Norman, Sierra, and Jennings [NSJ98] say that the absence of obligation not to do α should not imply permission to do α as is assumed in SDL. Rather, it may be that an agent or norm system expresses no attitude with respect to α. Norman et al. contend that an agent is permitted to perform an action only if the formula for permission is explicitly recorded, so absence of obligation not to perform does not imply permission to perform as in a negation-by-failure interpretation of SDL. Cholvy and Cuppens [CC98] comment that, in SDL, 'regulation r says that α is *not obligatory*' is not distinguishable from 'regulation r *does not say* that α is obligatory'. In the former case, it may be argued that there is an explicit permission (privilege) to do $\neg\alpha$. That is, in Andersonian semantics [And58], the former case says that refraining from doing α definitely does not bring about a violation. In the latter case, the regulation simply does not comment on whether or not refraining from doing α brings about a violation.

Requirement 31	Directed obligations, with actor, beneficiary, liable party, source utterance, and issuer, must be expressible.

SDL does not address *collective obligations* (page 42) [CP2000]. Here, some member of a group is expected to fulfil the obligation (e.g. 'some nurse must monitor the patient'), but there is not necessarily anyone that bears personal responsibility for the behaviour of the group.

2.6.4 Obligations are viewed as operators, not entities

SDL denotes obligations with the modal operator O, rather than identifying each obligation as an individual entity. Makinson [Mak99], following Alchourrón and Bulygin, suggests we need to distinguish between norms and propositions about norms. It is suggested that expressions such as $O(\alpha \rightarrow \beta)$, where truth values are taken as arguments to obligations, are troublesome to interpret as it is more natural to conceive obligations as related to described states-of-affairs, rather than to truth-values. Various recent authors have moved from operator-based accounts towards identified speech acts, such as identified obligations [Kim2001, AB2002b, AK2002]. A consequence of SDL's operator-based account is that particular obligations cannot be tagged with their current state, making the obligation life-cycle difficult to represent.

Requirement 32	Obligations should be individually identified.

2.6.5 Temporal aspects and lifetime of norms are not addressed

SDL is *static* [CDMR2001, Dam2002] and sets aside the dynamic *temporal aspects* of obligations. Taveter and Wagner [TW2001a, TW2001b, Wag2001] complain that SDL abstracts away from the current circumstances, which play an important role when dealing with deontic concepts in practice.

In terms of '*birth*' of norms, SDL does not model the *sources* of obligations in both norms and eventualities: for instance, it does not show how 'John must write his exams' (obligation) is derived from 'Students must write their exams' (norm) and 'John is a student' (eventuality). Dignum and Weigand [DW95] argue that little was said in early deontic approaches about how norms are established. They contend that deontic effects arise as a result of communicative actions. SDL omits treatment of *conditional obligations* [Che80, JS93], such as the obligation to fasten your seat belt when driving a car, which is incorrectly modelled by regular material implication [dAMW96]. Conditional operators [Che80] and dyadic deontic logics [PS97] investigate this.

In terms of '*death*' of norms, SDL does not model *punishment* and *reward*.

- SDL fails to model *punishment*: SDL leaves the violation and its consequences unspecified [TT99]. Anderson [And58] proposes that $O\alpha$ be reduced to $\Box(\neg\alpha \rightarrow V)$, meaning 'necessarily $\neg\alpha$ implies violation'. However, Anderson's violation atom, V, is vague as it speaks of the violation of the whole normative system, rather than showing exactly where violation has arisen [dAMW93]. Dignum and Kuiper [DK98] index the violation predicate into the action that caused the violation ($Violation(\alpha)$ for violation of $O\alpha$) but not to the violated provision. Dignum and Weigand [DW95] use the special predicate $Violation_{i,j}$ to model the violation of an agreement between agents *i* and *j*. This is unsatisfactory as it does not specify which aspect of the agreement was violated. d'Altan, Meyer, and Wieringa [dAMW93] propose the indexed atom $Violation_l$ to indicate the violation of a piece of legislation *l*,

such as a paragraph of a code of laws. This too is unsatisfactory as each paragraph may contain multiple obligations and prohibitions, and it is unclear which specifically was violated. Distinguishing between violation of different clauses is essential since violations of different clauses – and indeed even violations of the same clauses at different times – have different consequences. Observers typically wish to be aware not only of the fact that there was a violation, but also of the precise norm that was violated and when it was violated, so that they may assess the impact of the violation. They may wish to quantify the exact loss to the aggrieved party as a result of a particular violation at a particular time. Obliged parties face similar requirements: they may wish to choose which of a set of conflicting obligations to violate based on the expected costs of different potential violations. Both indexed atoms, $Violation_{i,j}$ and $Violation_l$, fail to differentiate between different violations of the same provision across time. Dignum and Kuiper's attempt to incorporate temporal aspects of violations [DK98] speaks of a semi-persistent violation predicate that 'disappears' after a special 'repair' action has been performed. This is problematic as violation history is not kept.

- SDL fails to model *reward*: Just as SDL does not model the punishment by omitting a treatment of *violation*, it also fails to model the reward by omitting treatment of *fulfilment*. That is, SDL does not model the legal consequences of keeping your obligations. Anderson's treatment [And58], is insufficient to model fulfilment as 'not-violated' is vague and may mean 'fulfilled' or 'not yet fulfilled/violated'.

SDL treats *standing* obligations (e.g. 'to honour your parents') but not *dynamic*, *dischargeable* obligations (e.g. 'to return the book') [Bro96, Bro2000]. Though it seems desirable in business to derive $\neg O\alpha$ from $O\alpha \wedge \alpha$ (since fulfilling a dischargeable obligation ends the obligation) the ideal worlds semantics of SDL treats obligations as long-lived ideals. Applying the *and-elimination* rule of (atemporal) propositional logic to $O\alpha \wedge \alpha$ yields $O\alpha$ as dynamic aspects are not treated.

The *ought implies can* rule (page 63), incorporated by some authors [Che80], needlessly encodes a rigid system of law for determining obligation life-cycle – Brown [Bro96] reminds that present inability to repay a debt does not render the obligation to repay null and void, as the obligation is typically sustained. The system of law may or may not provide that bankruptcy, disability, or death, render the obligation null – mere impossibility is not sufficient criterion for deciding on the status of the obligation, and the precise criteria differ across divergent systems of law.

Requirement 33	**Obligation life-cycle must be modelled. Obligations must be traceable to the events, states, and regulations that brought them about or cancelled them out.**

Cardinality of actions is an issue [CDMR2001]: permission to do an action in the standard approach is a permission to do the action many times. SDL only expresses standing permissions; permissions to do something once, or *n* times, cannot be stated.

Requirement 34	**Both once-off and persistent rights should be expressible.**

As SDL does not model change of legal relations, Jones and Sergot [JS96] supplement the *prescriptive* normative notions of deontic logic (obligations, prohibitions, and permissions) with the *descriptive* normative notion of power (legal capability) to bring about states of affairs. Immunity (legal disability of another party to bring about a state of affairs) must also be represented.

Requirement 35	**Primitives governing change of legal relations – such as power and immunity [Hoh78] – must be provided.**

In addition to putting aside the temporal nature of obligations themselves, SDL does not model the temporal nature of the *obliged actions*: being obliged to give a talk

today does not imply being obliged to give a talk tomorrow, even if the obligation to give the talk is not fulfilled [Kro97]. A talk on 21st October will not do if a talk on 20th October was what was required, since the former does not fit the 'quality criteria' for the desired occurrence. SDL omits consideration of action types [Wag2001], and other action quality criteria.

Various authors have investigated temporal deontic logics. Dignum et al. [DWV96, DK97, DK98] complain that Meyer's deontic dynamic logic is capable of expressing only immediate obligations where the action should be performed as the next action. They argue for an extension that caters for obligations to perform an action as soon as possible, before a deadline (relative or absolute time condition), or periodically. Examples of obligations specifying a deadline include obligations to 'pay within 30 days of purchasing', 'pass within 1 year of enrolling', 'order before stock is too low', or 'repair the roof before the October rains start'. Periodic obligations may include 'pay employees between 25th and 30th of every month' or 'order whenever available stock is between 6 and 10 units'. Dignum et al. model the dynamic system in terms of a purely theoretical formal mathematical Kripke structure, providing no practical implementation. Initially [DWV96], these structures constrained their framework by forcing the impractical assumption that all actions take the same amount of time. Later [DK97], they propose that an assumption of instantaneous begin- and end-actions may be appropriate, but requires further investigation.

More recently, Bons et al. [BDLT2000] introduce an 'epistemic dynamic deontic temporal logic', using the notation $[\beta]O\alpha$ to indicate that state-of-affairs β causes a transition to the state-of-affairs $O\alpha$ where α is obliged. As an example, $[promise(r1,r2,\alpha)]O_{r1r2}(\alpha)$ means that 'after *r1* promises to perform α to *r2*, then *r1* is obliged towards *r2* to actually perform the action α'. State history is not kept and contradictory rules and conflict resolution are not attended to. For instance, subsequent imposition of a law to the effect that 'only promises made by legally competent parties lead to obligations' could not be introduced.

Eiter et al. [ESP99] introduce a *do* operator for the absence of a mechanism for triggering state changes in standard deontics. Bons et al. [BDLT2000] proceed similarly; their *DO(α)* operator entails that α will hold at the next moment; and *DONE(α)* entails that α held at the very last moment (*α* was the very last action performed). For instance, *DONE(pay(John,$100))* indicates that the very last action was John paying $100. This operator is troublesome as it needs to be reset after each system event to reflect which action was last performed, and problematic race conditions can be expected in the determination of exactly which was the last action performed. No action history is maintained and logical errors arise as *DONE(α)* is overwritten.

2.6.6 Application of the theory

Minksy and Ungureanu [MU2000], citing Roscheisen and Winograd, remark that deontic logic allows one to reason about what an agent must do, but provides no means of ensuring that what needs to be done will actually be done – that is, it provides no direct help in the enforcement of obligations. Nevertheless, there are a vast number of projects which apply notions from deontic logic; see for example Meyer and Wieringa [MW93, pp. 17-40]. An illustrative sample of recent initiatives that have employed notions inspired by deontic logic is given in Table 2 below.

Application Area	Examples
Legal reasoning	**DX Deontic Expert System** [LR94] **Norman-G** [Ser2001]
Contract assessment	Daskalopulu [DDM2001, DM2001] Lee [BLWW95]
Analysis and modelling phase of system development	**Norm Analysis** [LO99, Liu2000, LSDM2001] **Agent-Object-Relationship modelling** [Wag2000, Wag2001]
Implementation techniques Trigger-based temporal database: Object-oriented code generation: Agent-oriented programming:	 **LEGOL** and **NormBase** [Liu2000] **COLOR-X** [BvdR95, DSvdR2000, SvdRD2000] **Knowledge-Perception-Memory-Commitment agents** [TW2001a, TW2001b]

Table 2: Sample of some recent applications of deontic logic

2.7 Conclusion: Requirements for a Solution

This chapter has reviewed the literature on event representation and monitoring, business rules, policy-based approaches, business process specification and automation, contracts, and deontic logic. From this review, we can derive the following criteria that need to be satisfied by a comprehensive e-commerce application development methodology and environment:

2.7.1 Store rich descriptions of business events and states (Chapter 3)

Events and states in commerce require a richer and more general representation than that provided by the event monitoring literature. A schema grounded in the philosophical writings, and corresponding services for the storage and interrogation of past events and states, is required. Business process automation would benefit from a generic occurrence monitor, which is able to assess which contractual provisions are relevant based on current events and states and past history. Dynamic classification, rather than static object-oriented class hierarchies, should be supported. This is because in legal environments the class of an item, and pertinent provisions, are determined by conformance of the item to an explicit description.

2.7.2 Support the transition from analysis to implementation (Chapter 4)

The primitives of conventional event-, policy-, workflow-, and contract-implementation languages are technical in nature and do not mirror the legal concepts present in business specifications. van Lamsweerde [Lam2001] suggests that software modelling research should propagate requirements abstractions down to the programming level rather than abstracting programming constructs up to requirements level as is conventional. An e-commerce application development

method should give guidance to aid the transition from analysis to implementation. Human analysts would benefit from suggestions for exposing implementation primitives from a principled analysis of specifications and contracts. Whilst the difference between natural language policy statements and lower-level machine-readable policies is widely recognized (§2.3.3), little has been done to support the translation. Controlled languages for executable specification such as ACE (page 49) fall short of the ability to model contracts. An important aim is to facilitate the refinement of an unstructured specification into a format amenable to direct analysis and implementation. The methodology must provide guidelines to assist in uncovering the formal structure underlying specifications of contracts, policies, and regulations written in English.

2.7.3 Model and store legal provisions (Chapter 5)

Deontic logicians and jurisprudential theorists have long advocated the need for representation of obligations, permissions and prohibitions [vW51, And58, And62, MW93], and institutional powers [Hoh78, JS96, JS2000]. Advanced e-commerce support requires incorporation of deontic and jurisprudential primitives to model provisions in terms of fundamental legal conceptions. These primitives will provide *compositionality* – the ability to build sophisticated legal systems from basic blocks, rather than hard-coding the assumptions of any one system of law.

The business process must be guided by a dynamically changing set of contractual and regulatory provisions. Approaches based on standard form contracts and contract templates are too rigid. Once-off validation of contracts at contract creation time is insufficient (see page 56) because recently added policies and legislation may affect provision validity.

Major normative notions include the representation of ought-to-do and ought-to-be obligations; personal and impersonal obligations; prima facie (defeasible) obligations; conditional, and secondary obligations; legal powers and immunities, prohibitions and permissions; one-shot and persistent rights; obligations over time; and violations. A software implementation of contracts and regulations should

therefore incorporate these notions, and be capable of expressing both prescriptive and descriptive norms, drawn from intra-, inter-, and extra-organizational policies.

2.7.4 Express, detect, and resolve conflict (Chapter 6)

The inevitable tendency for business specifications to contain multiple contradictory provisions pertaining to a given circumstance indicates the need for sophisticated facilities both to detect conflicts at the time provisions are added to the data store, and to resolve conflicts at run-time. Conflict resolution should not merely be by priority numbering and implicit text ordering of rules. Advanced facilities should be able to determine which legal consequences of provisions are null in a situation, by which laws, and why.

2.7.5 Monitor and enforce provisions (Chapter 7)

A major goal of this research is substantive progress towards machine-checkable and executable contracts. An architecture, including appropriate mechanisms and interfaces, for efficiently monitoring and enforcing contracts must be provided.

The chapters that follow introduce novel techniques and tools for addressing these requirements.

Chapter 3

Occurrences in Electronic Commerce

This chapter describes an approach for representing and reasoning about occurrences in electronic commerce.

We first define what is meant by an occurrence and demonstrate how an occurrence may be used to store a workflow event, such as a payment, or an association state, such as 'being a supplier for' (§3.1). We continue with an explanation of various types of queries and demonstrate how queries can be represented and stored (§3.2). We shall see later (Chapter 5) that the storage of queries provides us with the ability to store provisions, which are propositions with nested propositional content. Query storage is necessary in order to determine which recorded descriptions describe a given occurrence or item, as well as which recorded descriptions overlap: processes we call 'coverage checking' (§3.3). The ability to determine which descriptions (queries) cover an occurrence or item allows us to determine which provisions apply to that occurrence or item (Chapter 7). Furthermore, the facility to dynamically detect overlaps between descriptions enables conflict detection (Chapter 6).

3.1 Representing and Storing Occurrences

We treat an **occurrence** as being an instance of a specific relationship type or association type that exists between entities, at a moment in time or over an interval in time. For instance, we would treat *buying*, *owning*, *approving*, *being-obliged*, and *being-prohibited* as occurrence types. Each occurrence has a single type and an occurrence type (such as *buying*) may have multiple occurrence instances. Each occurrence instance has role-players acting in a role in the occurrence: an occurrence of *buying* typically has at least participants in the roles *buyer*, *seller*, *sold item*, and *purchase price*[7]. An occurrence may be an event, a state, or a process. Consider an event of *supplying* that is instantaneous, and has as participants a supplier, a supplied party, and an item supplied at that instant. In contrast, a state of *being supplier* has (perhaps unspecified) duration, where the supplier and supplied party have an association for the whole duration of the state. And a process of *supplying* does not imply that what is being supplied is ever supplied. When referring to an occurrence we should generally clarify which semantics (event, state, or process) we mean, and also which sense of the word we intend.

Different occurrences may overlap in time: John may be involved in a process of buying a fridge, while Jeff is involved in a process of buying a car. Indeed, if both processes are brought to a successful conclusion by the single fall of a hammer in a clearance sale, then the occurrence of the instantaneous events of buying could happen simultaneously as well. For this reason, occurrence times are insufficient to identify occurrences, and unique identifiers are used to distinguish occurrences.

We have chosen the term 'occurrence' since the term 'event' used in the philosophical literature (page 16) to describe a momentary or prolonged state of affairs, is typically understood as being instantaneous in the active database and systems programming literature (page 15). Furthermore, we do not wish to imply

[7] Inanimate 'participants', such as *purchase price*, might be more easily thought of as attribute values for attributes of the occurrence.

that our notion shares any of the philosophical subtleties of various uses of the term 'event' in the literature. For instance, Bennett [Ben88], controversially (though he is not alone) argues that John's crossing the Channel and John's swimming the Channel are the same 'event', whereas we treat them as separate occurrences. Our notion of an 'occurrence' most closely resembles Parsons' notion of an 'eventuality' (an event, state, or process), which Kimbrough [Kim2001] has applied to deontic reasoning.

☑
Requirement 1 (pg 19): The information model must be able to describe occurrences of past events, states, and processes, as well as the (active and passive) role-players involved.

We treat occurrences of events, states, and processes uniformly in certain respects. Our contention is that all indicate a temporal relatedness between participants in various roles, and the abstract notion of an 'occurrence' is therefore useful. Policies such as 'Bob is forbidden to *buy* X' or 'Bob is forbidden to *own* X' speak of occurrences of *buying* and *owning* in pragmatically similar ways even though the former seems to be forbidding the occurrence of an event and the latter forbidding the occurrence of a state. As we saw earlier (page 42) high-level policies must be able to speak of either actions or states of affairs: this requirement can be met using an approach based on occurrences, which provide a uniform treatment of events and states for obligation fulfilment monitoring. For our purposes we wish to exploit the similarity in order to uniformly monitor for the occurrence of events, states, and processes over time. Furthermore, we can provide a uniform set of facilities for querying occurrences, and their role-holders, without needing to explicitly decide in each case whether the occurrence is instantaneous or prolonged.

☑
Requirement 4 (pg 22): Persistent storage of business-level occurrences is required.

Let us illustrate with an example. Take the scenario, from §1.2, where Steelmans is a supplier for SkyHi, and SkyHi has paid them $25,000 for a specific delivery. Ignore for the moment any contractual relationships (contract-related occurrences), which we return to later, and consider only the simple occurrence of a state of being a supplier, and an occurrence of an event of paying. Let being_supplier1 and paying1 be Skolem constants that name

occurrence instances of types *being a supplier* and *paying* respectively. We might add allocating1 to denote that the payment was *allocated*, using the First-In First-Out basis, to a delivery, and occurring1 to indicate that the payment *occurred* on 15th August 2001. A basic occurrence-role-participant tabular schema can be employed for storing occurrences: in each row, we store the occurrence identifier and a participant and its role in the occurrence. Table 3 depicts a tabular representation of the above-mentioned occurrences. For readability we have included values like Steelmans in our tables instead of foreign key references. Similarly we show occurrence primary keys in forms such as being_supplier1, instead of foreign key references into a table describing the occurrence type *being_supplier*. Finally we omit repeated key values in adjacent rows.

☑

Requirement 6 (pg 24): A business-level, rather than technical-system-level, approach is needed for business process automation.

Commentary	Occurrence	Role	Participant
Steelmans *being a supplier* for SkyHi	being_supplier1	supplier	Steelmans
		supplied	SkyHi
According to Clause 1, SkyHi *paid* $25,000 to Steelmans	paying1	payer	SkyHi
		paid-amount	$25,000
		payee	Steelmans
		isAccordingTo	Clause 1
The payment was *allocated* to a previous delivery	allocating1	allocated	paying1
		allocatedTo	delivering1
		allocationBasis	FIFO
The payment *occurred* on 15 August 2001	occurring1	occurred	paying1
		occurredOn	15 August 2001

Table 3: A tabular schema for storing various occurrences

The storage schema depicted above bears some resemblance to the graph-based binary-relational model used by Ayres and King in the functional database language **Hydra** [AK96], and shares some of the same useful properties. Ayres and King

comment that conventional relational and object-oriented database systems are unable to support associational queries corresponding to questions such as 'what is the relationship between Sandra and Mike?'. Hydra, in contrast, provides associational primitives, which return the list of functions or inverse-functions that relate two items. Our occurrence-centric representation provides similar advantages, facilitating queries such as '(select occurrences in which Steelmans participates) intersection (select occurrences where SkyHi participates)'. This would return paying1 and being_supplier1 in the example above. This style of associational query is especially useful in commerce applications for determining legal relationships between parties. For instance, an associational query may be framed to determine the history of interactions between two parties, or indeed to determine what obligations (occurrences of being-obliged) bind two parties.

Comparing our approach to Kimbrough's ESΘ theory (op cit, page 17), Kimbrough represents events and states using event variables and either thematic or grammatical roles. An occurrence of SkyHi paying $25,000 to Steelmans for a particular delivery would be formally represented as:

$$\exists e \, (\, paying(e) \wedge Subject(e,SkyHi) \wedge DirectObject(e,\$25000) \wedge \\ IndirectObject(e,Steelmans) \wedge Sake(e,e') \wedge delivering(e') \,)$$

Formula 1

Our 'occurrences' differ from Kimbrough's events and states in the following ways:

- we choose to use domain- or application-specific roles – so-called 'deep roles' [JM2000] – rather than generic thematic roles (such as agent, theme, etc.) or grammatical syntax markers (such as subject, object, etc.), in the representation of occurrences. While generic thematic roles – such as agent, patient, instrument, source, and destination – are often helpful and are commonly used in knowledge representation in artificial intelligence [All95, Gri94, Sow2000], they have been criticized. Davis [Dav96], for instance, argues that the thematic role of a participant in an event occurrence may be difficult to determine or ambiguous. Citing Jackendoff, Fillmore, Gawron, and Croft, Davis explains that in a

commercial event such as *buying* both the buyer and the seller can be construed to be in the agent thematic role, as the buyer acts to supply money and the seller acts to supply goods. Using domain-specific roles (such as buyer, seller, etc.) allows us to avoid this vagueness. An alternative to thematic roles is the use of grammatical syntax markers such as subject and object. Kimbrough, acknowledging that it is almost certainly preferable to employ semantic markers instead, uses these for illustration purposes only. Grammatical syntax markers have vague semantics: consider that, in 'John opened the door' (active or causative reading) and 'the door opened' (passive or inchoative reading), the door occupies different syntactic positions – object and subject respectively – even though its semantic role as being opened stays constant. One of the triumphs of event semantics is semantic stability in its treatment of active and passive sentences [Par90]; a benefit lost by simplifying the illustration with grammatical syntax markers. We prefer to employ domain-specific roles, such as opener and opened, as these are stable, precise, and simple for an analyst untrained in linguistics to identify and comprehend.

- to support measures by different parties over time we employ measuring occurrences (see §4.3) instead of Kimbrough's *unit()* and *quantity()* predicates [Kim98a].
- we employ allocating occurrences (see §5.6.1) in place of Kimbrough's *Sake(…)* predicate to allow different parties to allocate performances to obligations using different bases.
- we employ a tabular representation, which we believe improves accessibility to the set of roles for an occurrence, enabling them to be easily stored and queried using conventional relational query languages, such as SQL. It may be difficult to retrieve the set of roles for a Kimbrian occurrence in a logic program, since some logic programming implementations are not capable of resolving X(e1,Y), to a set of predicates (roles), X.

☑

Requirement 5 (pg 23): Business applications require occurrences to be accessible via ad-hoc queries, rather than via fixed access paths.

In the case of simple occurrences, the representation we offer (tuples versus Kimbrough's logical expressions) affords us an implementation mechanism using relational, or just tabular, data stores. The representation is similar whether the occurrences are stored in a relational database, held in a spreadsheet, or transmitted in a comma-delimited or XML text file. No data transposition is technically necessary, so streaming the data into and out of the database from communication channels can proceed without schema transformation difficulties, and using very simple and efficient parsers. We have not dealt with compression or data clustering techniques, though it seems clear the representation could be stored and accessed more efficiently without loss of information.

In occurrences that are more complex and nest propositional content, the relationship to Kimbrough's Disquotation operator (op cit, page 17) is more removed, but, we believe, still faithful to the intentions of the logic. The details are described in Chapter 5. In spite of the additional complexity of occurrences which nest propositional content (i.e. queries), the same simple tabular structure still suffices for representation purposes and no changes to the data schema, transmission format, or parsers are required.

3.2 Representing and Storing Queries

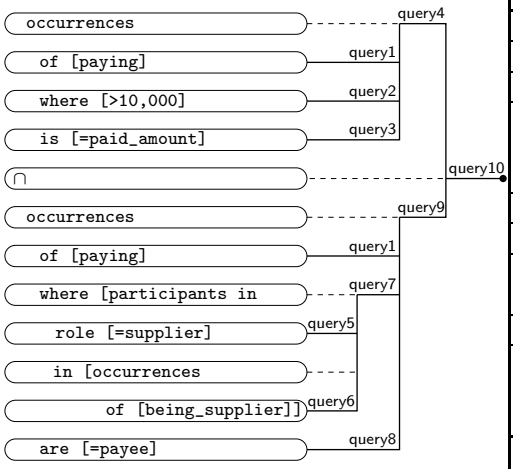
In the previous section, participants were denoted extensionally: that is, by indicating the specific identifier of the participant in the occurrence. It is also possible to select participants intensionally, by providing a description of the participants; all identifiers that fit the description at a particular time are then viewed as participating in the occurrence at that time. A description can be viewed as a query that defines a set of criteria and returns a set of resulting identifiers, which may vary over time as the contents on the knowledge-base changes. The benefit of using descriptions, instead of identifiers, is in enhanced **stability** of the specification. An obligation of John to process an insurance claim would become meaningless if John left the company. But an obligation of the claims processing clerk to process an insurance claim would

ensure that Mary assumes responsibility for processing an insurance claim as she takes over John's position as claims processing clerk. In the latter case, the query 'claims processing clerk' returns first John and then, as human resources in the organization change, returns Mary in his place.

For the purposes of representing policies and contracts and determining which provisions apply (Chapter 5), queries need to be stored rather than merely answered and discarded as is typically the case with conventional relational databases. A named (or identified) query is often called a *view* in the database literature. In our case, given the potentially large number of identified queries (views) we need to record, we must store them in a database, rather than merely hold them in memory.

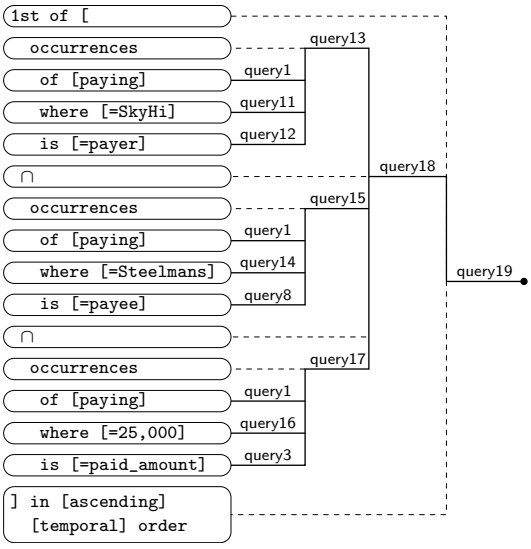
Queries can be stored by storing their identifiers, and values for their criteria. For this purpose we assign names to each type of criterion. Depending on the type of query, the criterion-value for a query may be either a constant value or another query. Queries may be stored in occurrence-role-participant tabular form by assigning a query-occurrence-identifier, and, for each criterion storing the criterion-value in the participant column, and the criterion-type in the role-column. The criterion-value may be a constant or a reference to an embedded query. EDEE's parser takes the textual form of the query in our purpose-built language, EDEEQL, and converts it to its tabular semantic form. EDEEQL currently defines algebraic, alphabetic, set-theoretic, occurrence-related, ordinal, and nested queries. See Appendix 1 for further details of the syntax and storage schemata of these query types.

Take for example the query that returns all occurrences where more than $10,000 is paid to a supplier. The complete query, identified as Query10, is shown in Figure 2 below, which illustrates the parse tree for the query. Similarly, we can store another query, Query19, that returns the first occurrence of SkyHi paying Steelmans $25,000 for the delivery before 1 September 2001. The parse tree and tabular schema for Query19 is shown in Figure 3 below.



QueryID	Criterion Type	Value
query1	type	paying
query2	greater-than	10000
query3	identified-concept	paid_amount
query4	occurrence	query1
	participant	query2
	role	query3
query5	identified-concept	supplier
query6	type	being_supplier
query7	role	query5
	occurrence	query6
query8	identified-concept	payee
query9	occurrence	query1
	participant	query7
	role	query8
query10	intersectand	query4
	intersectand	query9

Figure 2: Parse tree and storage schema for a query that returns all occurrences where more than $10,000 is paid to a supplier



QueryID	Criterion-type	Value
query11	identified-concept	SkyHi
query12	identified-concept	payer
query13	occurrence	query1
	participant	query11
	role	query12
query14	identified-concept	Steelmans
query15	occurrence	query1
	participant	query14
	role	query8
query16	equal-to	25000
query17	occurrence	query1
	participant	query16
	role	query3
query18	intersectand	query13
	intersectand	query15
	intersectand	query17
query19	set	query18

Figure 3: Parse tree and storage schema for a query that returns the first payment of $25,000 by SkyHi to Steelmans

The ability to store queries also means that we can use queries to look-up other (stored) queries. The ability to look-up queries that return results which fit certain criteria is useful for finding which queries cover a certain item. This in turn is useful for determining which policies cover a particular item, and for interrogating a

contract to determine what items and eventualities the contract covers. We describe the mechanism for finding covering-queries, and its use in finding the policies applicable to an item, in more detail in the next sub-section (§3.3).

3.3 Determining Covering-Queries

Finding covering-queries is the reverse of query resolution. In query resolution, we begin with a defined query and return all the results that match the specified criteria. When finding covering-queries we begin with an item, and determine which queries cover the item. The item may be a regular concept or it may itself be a query, in which case we can still determine analytically which other queries completely cover the results of that query. The process is analogous to the process of determining which subscriptions cover an event, described in the literature on publish-subscribe event notification (§2.1). As we saw earlier though, for contracts we require persistent events, explicitly stored subscriptions, and traceable inferences. We demonstrate here, and in Chapter 5, how a novel continuous query mechanism based on stored queries addresses these requirements.

3.3.1 Overview of coverage checking

Figure 4 below illustrates the complementary nature of query resolution and determination of covering-queries. Assume we have the occurrences 'SkyHi paying Steelmans $25,000' (paying1), 'SkyHi paying BrickMen $5,000' (paying2), and 'SkyHi paying their employee, Jack, $15,000' (paying3) stored in the occurrence store. Further, assume we have the queries 'payments of more than $10,000' (query4, in Figure 2 above) and 'payments to suppliers' (query9, in Figure 2 above) stored in the occurrence store. Turning now to **query resolution**, the query 'payments to suppliers' would return the occurrences paying1 and paying2. Similarly, the query 'payments of more than $10,000' would return the occurrences paying1 and paying3. If we consider the reverse, **determination of covering-queries**, we see that the occurrence paying1 is covered by both the queries 'payments of more than $10,000' (query4) and 'payments

to suppliers' (query9). We say that an occurrence (or indeed any item) *fits a description* if it is *covered by a stored query*.

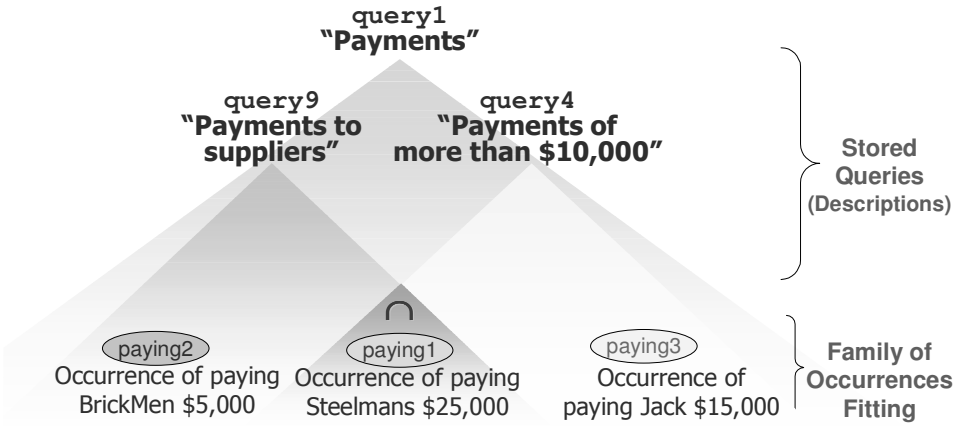


Figure 4: Occurrences fitting a description (covered by a stored query)

The notions of covering relationships between queries, and dirtying relationships between data and queries, are used to find run-time overlaps. We say that a query is *covered* by another stored query if the results of the former are a subset of the results of the latter for any data-set. Some questions of coverage are decidable statically, but others depend on application semantics: some covering relations change when new data is added, in a context-specific manner. We say a query is *dirtied* by new data (*input dirt*) if the new data changes a *criterion* (cf. text of a WHERE clause in an SQL SELECT statement) of the query. For example, upon the addition of the new supplier, Steelmans, to the database, the query 'payments to **suppliers**' is dirtied as the results must now also include any 'payments to **Steelmans**'. Any such payments would be what we term *output dirt*. The materialized view literature [GM95] talks of dirt in the sense of our output dirt. Whereas materialized views would only change when any actual payments to Steelmans were added, covering relationships may change even in the absence of any payments stored in the database.

We now look at detailed worked examples that show how we determine which queries cover an item, and which queries cover a query.

3.3.2 Worked example 1: Queries covering *items*

Take the query 'payments of more than $10,000 to suppliers' (Query10 in Figure 2 on page 85). Assume that this query is stored in an empty database. Then, assume that we record, in this database, that Steelmans is a supplier of SkyHi, by inserting an occurrence, being_supplier1, as described in Table 3 on page 80. (Assume that the payment, paying1, is not inserted yet and is only inserted later.) Upon insertion of the rows for being_supplier1 the coverage-checking algorithm examines each of the unique items – Steelmans, being_supplier1, supplier, SkyHi, and supplied – in the set of triples added to the database for the occurrence:

1.	*By Rule A2.2.10*[8]*,*	the query `[=supplier]` (Query5) covers supplier
2.	*By Rule A2.2.2,*	item being_supplier1 is covered by the query `occurrences of [being_supplier]` (Query6)
3.	*By Rule A2.3.2,*	queries `[=supplier]` (Query5) and `occurrences of [being_supplier]` (Query6) dirty `[participants in role [=supplier] in [occurrences of [being_supplier]]]` (Query7). Substituting the input dirt (shown underlined) for the dirtied criteria yields the partial re-evaluation query: `[participants in role [=supplier] in [=being_supplier1]]`. Evaluation of this partial re-evaluation query yields the output dirt Steelmans.
4.	*By Rule A2.3.2, and step 3*	The dirt, Steelmans, from the previous step dirties `occurrences of [paying] where [participants in role [=supplier] in [occurrences of [being_supplier]]] are [=payee]` (Query9). Substituting the input dirt (shown underlined) for the dirtied criterion yields the partial re-evaluation query: `occurrences of [paying] where [=Steelmans] are [=payee]`. Evaluation of this partial re-evaluation query yields no output dirt. The coverage-checker thus stops.

[8] Each of the rules mentioned here is defined in detail in Appendix 2, which defines the full set of coverage-checking rules, extracted from the coverage-checker module of our EDEE prototype.

We conclude that the new occurrence, being_supplier1, is covered only by the query `occurrences of [being_supplier]` (Query6). We record the queries that have been dirtied by this new data and cache the output dirt, since we can use the output dirt for partial re-evaluation of queries later. The list of dirtied queries and their output dirt is shown in Table 4 below. Though in this case only one item of dirt is produced for each query, the number of items of dirt may be greater than one. For instance, the addition of a new occurrence of being a supplier may cause a new supplier to be added as output dirt for query7 in Table 4 below.

☑

Requirement 17 (pg 38):
It must be possible to define and store the criteria that an item must satisfy – the description (query) it must fit – in order for it to count as being of a certain type.

Coverer (Dirtied Query)	Covered (Output Dirt)
query5	supplier
query6	being_supplier1
query7	Steelmans

query5 = `[=supplier]`

query6 = `[occurrences of [being_supplier]]`

query7 = `[participants in the role [=supplier] in [occurrences of [being_supplier]]]`

(see Figure 2 on page 85 for full definitions of each of these queries)

Table 4: Dirtied queries and their output dirt, stored in the `EdeeCoverer` table, after addition of the occurrence being_supplier1 to a new datastore

By a similar coverage-determination process we can coverage-check items such as Steelmans in order to answer such queries as 'which provisions pertain to Steelmans?', even as the number and values of Steelmans attributes (i.e. the occurrences associated with Steelmans) change over time. The representation of provisions using nested queries is dealt with in Chapter 5.

☑

Requirement 18 (pg 39):
We must be able to express policies applying to intensionally described (rather than merely extensionally listed) groups of objects. The members of such groups may change dynamically.

The mechanism explained above demonstrates how we can incrementally determine which queries cover a newly added occurrence, or begin to cover an entity

whose attributes are dynamically changing. We can consequently determine (Chapter 7) which provisions cover an occurrence or mention an entity.

☑ **Requirement 3 (pg 22):** Matching (occurrence detection) must be against a long history.

The incremental nature of the algorithm is important as tens of thousands of occurrences and entities may be stored in the database. Re-executing every query stored in the database each time an occurrence is added is not feasible, particularly since most results will be unchanged. It is important therefore to re-execute only queries whose results may have been changed. The cache of dirtied queries provides a potential performance boost by allowing the creation of more specific partial re-evaluation queries. Even if the actual dirt cache was cleared to conserve resources, we can still rely upon the query optimizer to re-evaluate only the minimal set requiring re-evaluation. For large data volumes, a query execution approach is likely to be more efficient than a theorem-proving or logic programming approach, as the query execution approach incorporates query optimizers which take into account data profiles (predicate selectivity) when executing a query, whereas theorem provers and logic programs typically do not concern themselves with such execution efficiency issues.

3.3.3 Worked example 2: Queries covering *queries*

☑ **Requirement 10 (pg 30):** Pattern-pattern matching functionality is required for analytic policy conflict detection.

The determination by the coverage-checking algorithm of which queries cover a query, is also best illustrated through an example. Let us continue using the data store as it stands at the end of the previous example (§3.3.2). Assume that we record the query 'the first payment of $25,000 by SkyHi to Steelmans' (Query19 in Figure 3 on page 85), in the data store. Notice that, by Rule A2.2.3, `[=paid-amount]` (nested in Query19) is given the identifier Query3, as such a query is already stored in the parse tree of Query10. Similarly `[=payee]` (nested in Query19) is given the identifier Query8. Now, comparing Query19 to other queries stored in the database we find:

1.	*By Rule A2.2.9,*	`[=25,000]` (Query16) is covered by query `[>10,000]` (Query2)
2.	*By Rule A2.2.16 and step 1,*	`occurrences of [paying] where [>10,000] is [=paid_amount]` (Query4) covers `occurrences of [paying] where [=25,000] is [=paid-amount]` (Query17).
3.	*By Rule A2.2.11,*	`[=Steelmans]` (Query14) is covered by any query which covers Steelmans. As seen earlier, `[participants in the role [=supplier] in [occurrences of [being_supplier]]]` (Query7) covers Steelmans. This fact is stored in the last row of the 'dirtied query and dirt' cache shown in Table 4 (page 89) above. Therefore Query7 covers Query14.
4.	*By Rule A2.2.16 and step 3,*	The query `occurrences of [paying] where [participants in the role [=supplier] in [occurrences of [being_supplier]]] are [=payee]` (Query9) covers `occurrences of [paying] where [=Steelmans] is [=payee]` (Query15).
5.	*By Rule A2.2.14, step 2 and step 4,*	Query18 is covered by Query10.
6.	*By Rule A2.2.18 and step 5,*	The set criterion (Query18) covers Query19.
7.	*By Rule A2.2.5, step 5 and step 6,*	Query19 is covered by Query10.

This example shows how overlaps may be detected at the time queries are added to the database, or when inserted data dirties queries and thus brings queries into overlap; that is, **dynamically appearing overlaps** are detectable. In this case, the addition of the application data, being_supplier1, which says that Steelmans is a supplier for SkyHi, brought the queries 'payments of more than $10,000 to suppliers' and 'the first payment of $25,000 by SkyHi to Steelmans' into overlap at run-time.

☑

Requirement 7 (pg 28):
Situations should be interpreted against a dynamic set of rules.

3.3.4 Static and dynamic overlap

Determination of the relationships between queries may be performed at two times:

- ***statically*: at the time the query is added to the database.**

  Static relationships are analytic relationships that must hold between queries irrespective of the data in the database. For instance, it is statically determinable that the query `[occurrences of [paying] where [=SkyHi] is [=payer]] ∩ [occurrences of [paying] where [=25,000] is [=paid_amount]]` is covered by the query `occurrences of [paying] where [=SkyHi] is [=payer]` irrespective of what payments are actually stored in the database. For performance reasons, it is preferable to store static relationships at the time the query is added to the database, as these results can be derived once and used repeatedly.

- ***dynamically*: at the time occurrences are added to the databases.**

  Dynamic relationships are those that hold between queries as a result of the particular contents of the database. §3.3.3 gave an example of the determination of a dynamic relationship between queries. Because dynamic relationships change as data is added, they must be computed each time new occurrences are added to the database if immediate detection is required. Using actual results produced by EDEE's `CoverageChecker` class, Figure 5 and Figure 6 show how the addition of application data may change the covering relations between queries: compare the coverage graphs *before* and *after* the addition of being_supplier1 to the occurrence store.

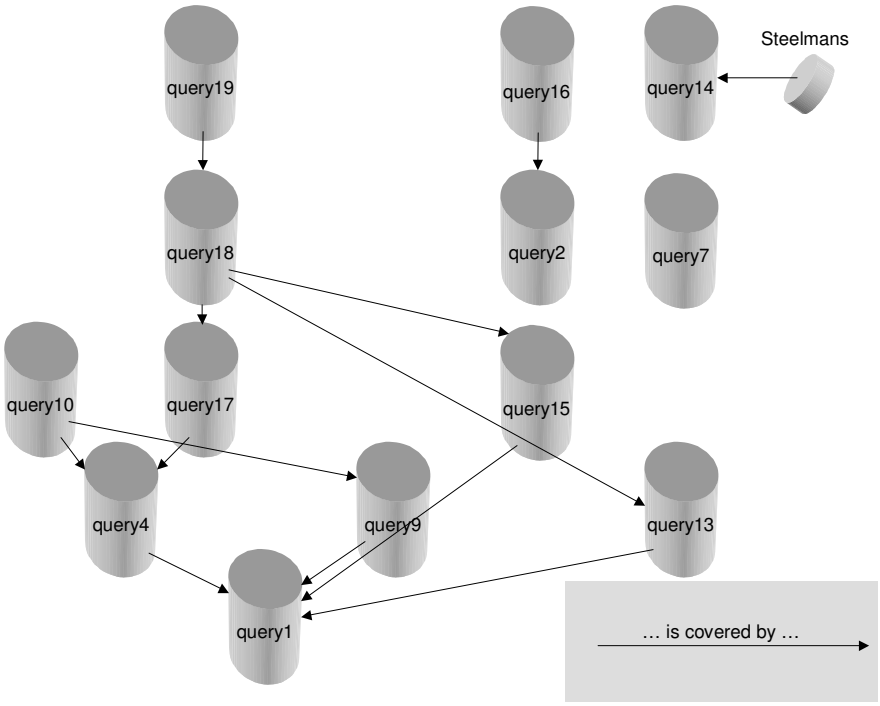


Figure 5: Covering relations graph *before* addition of being_supplier1

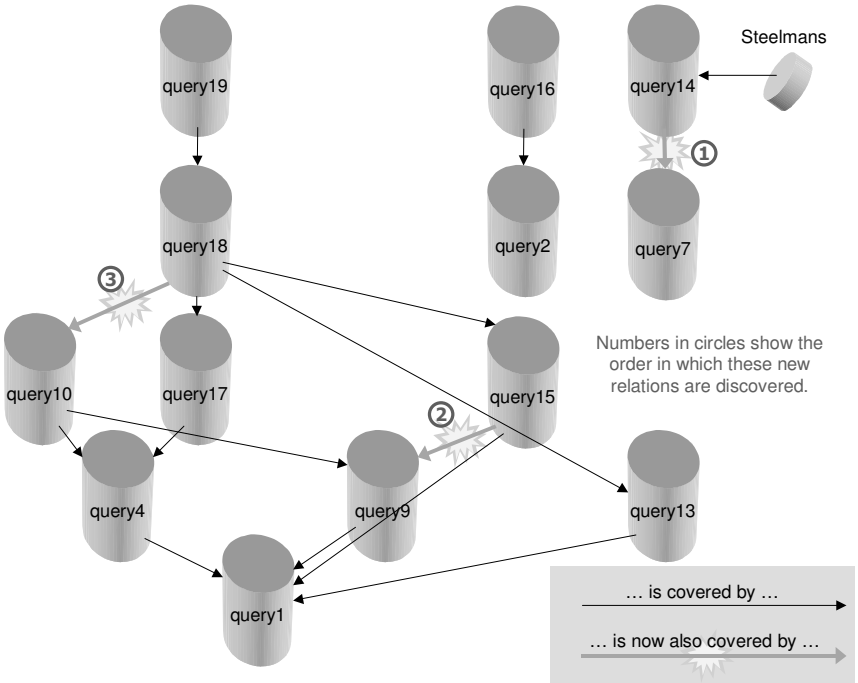


Figure 6: Covering relations graph *after* addition of being_supplier1

3.3.5 Applications of coverage checking

As we shall shortly see (Chapter 5) the ability to determine whether an occurrence (or other entity) is covered by a stored query allows us to rapidly *ascertain which of a large number of recorded provisions apply* to a given eventuality. The same facilities are helpful during *management review of policies*, as management can quickly determine which policies apply to a given user or class of user, to a given item or type of item (e.g. inventory item, machine, building, vehicle, or other asset), to a given location, or even to a given document.

In Chapter 6 we see that the ability to analytically spot overlap between queries facilitates *conflict detection* as co-applicable clauses with conflicting opinions can be found.

3.4 Summary

This chapter has tackled the first of the requirements identified by our critique of related work in the previous chapter.

We have reviewed the *representation and storage of occurrences* of instantaneous events and prolonged states. We have described a scheme for representing variable-attribute commercial events and states using an abstraction known as the occurrence. Occurrences have participants acting in various roles. The participants in an occurrence may be explicitly specified using an identifier, or may be described using a query. A persistent history of business eventualities (past events and states) can be maintained and interrogated using ad-hoc queries in SQL or EDEEQL. In our example (Table 3, page 80), occurrences of being_supplier and paying were stored.

Occurrences and other entities can be described using *queries*. We have described how queries may be stored and resolved, and we have shown in detail how we can determine which queries dynamically begin (or cease) to *cover* an item or another query. Our occurrence detection algorithm is able to detect even occurrences in the

distant past – e.g. that were added to the database many months earlier – but that are still pertinent to current occurrence evaluation.

Figure 7 summarises the data model employed in EDEE.

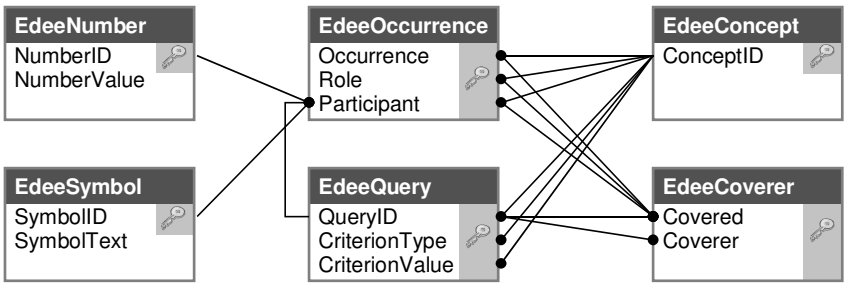


Figure 7: Data model used in EDEE

In the next chapter, we look at pointers that a business analyst can use to uncover the occurrence structure encoded in English language specifications of business process applications.

Chapter 4

From Analysis to Implementation

Some assistance is required in the process of transition from specification to implementation. The transition cannot be completely systematized; the natural language processing and artificial intelligence problem is, at present, simply too difficult [OM96]. We do not therefore propose to address natural language processing issues. Rather, we seek to pay some attention to how contract structure and workflow occurrences are expressed in natural language, with the goal of providing helpful insights to aid in capturing the essentials of a business specification. Previous work in the area of natural language requirements analysis includes the **KISS** approach and associated **Grammalizer** tool [HvdVH97], which help analysts to derive a conceptual model from a textual domain description.

This chapter details a set of guidelines that can be used by an analyst to undertake a formal analysis of business contracts and user requirements specifications. It outlines a set of basic rules that may be used to *expose occurrences* from appearances of certain words and word forms in English-language specifications. The guidelines are labelled and applied to worked examples from our application scenario (§1.2). The envisaged output of applying these rules is not procedural code, but rather contract structure that is actionable and can be

monitored. The analysis output can be used for direct software implementation, as the occurrences identified by the analyst can be stored in the occurrence store. While this is only a limited step, not equivalent to writing a whole application, it moves us towards the ultimate goal of driving applications from contracts.

In each of the sections that follow, we propose various techniques that can be employed to expose occurrences in business process application specifications. We look, in turn, at domain-specific occurrences (§4.1), and occurrences of selection (§4.2), quantification (§4.3), sorting and comparison (§4.4), prescription (§4.5), and description (§4.6).

4.1 Domain-Specific Occurrences

Identification of domain-specific occurrences from an English-language specification may proceed through a search for verbs, deverbative nouns, and roles, which indicate underlying occurrences.

Explicit verbs may indicate occurrences of events, states, or processes. Explicit verbs can be detected through a number of means including indicative suffixes, or consultation of a lexicon.

Indicative suffixes such as –ing, –s, and –ed on words often point to the existence of occurrences whose type is the gerund[9] form of the word. Occurrences are also often indicated by non-modal auxiliaries: 'is', 'was', 'being', 'been', 'are', 'were', 'will', 'have', 'has'.	**Guideline 1**

[9] A canonical form ending in –ing.

Examples:

By Guideline 1, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Occurrences of Type
owning		owning
owns		"
owned		"
is/has overdrawn		being overdrawn / overdrawing

Guidelines are applied as for any other rule defined in this thesis: evidence is taken in conjunction with the rule to justify a (prima facie) conclusion in terms of the rule. For example:

Evidence: Appearance of 'owned' in the English-language specification, …
Provenance: … according to **Guideline 1** above, …
Consequence: … indicates, an occurrence, or occurrences, of type *owning*.

Deverbative nouns are noun forms of verbs, and as such may also reveal underlying occurrences.

Indicative suffixes such as as -ion, -ment, -ent, -ure, -ance, -ence, -ancy, -ency, -ing, -al, -y, or –age in a deverbative noun often point to the existence of occurrences whose type is the gerund form of the deverbative noun.	**Guideline 2**

Examples:

By Guideline 2, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Occurrences of Type
registration		registering
acceptance		accepting
approval		approving
delivery		delivering
breakage		breaking
failure		failing

Examples from Application Scenario (§1.2):

tolerance	(Clause D.1)	tolerating
payment	(Clause P.3)	paying (Table 3, p80)
violation	(Clause L.3)	violating (Table 13, p152)
instigation	(")	instigating (")
compensation	(")	compensating
obligation	(")	being obliged (Table 11, p143)

Parsons (op cit, page 17), in his theory of event and state semantics, advises that some suffixes may be indicative of instances of states.

Suffixes such as such as –ness, -ship, –hood, and –ly may indicate underlying occurrences of states or events.	**Guideline 3**

Examples:

By Guideline 3, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Occurrences of Type
illness		being ill (state)
membership		being a member (state)
allegedly		alleging (event)

Examples from Application Scenario (§1.2):

thickness	(Clause D.1)	being thick
More specifically, 'being thick', indicates an occurrence of measuring where dimension_measured is 'thickness' (see §4.3)		

There are also a number of ways in which occurrences of *possessing* seem to be indicated in English.

The ending –s' or –'s, the preposition 'of', and the possessive pronouns 'his', 'her', 'their', 'our', and 'its' may indicate occurrences of *possessing*, *having*, or *owning*.	**Guideline 4**

Word forms denoting the names of **roles** held by participants in an occurrence often indicate underlying domain-specific occurrences.

Indicative suffixes such as –er, –or, –ar, –ee, –ant, –ent, –ed, -d, –en, or –yst on an English word typically denote role names, and often point to the existence of occurrences whose type is the gerund form of the role name.	**Guideline 5**

Examples:

By Guideline 5, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Occurrences of Type
employer		employing
director		directing
registrar		registering
payee		paying
applicant		applying
resident		residing
delivered		delivering
forbidden		forbidding
analyst		analysing

Examples from Application Scenario:

paid	(Clause C.4)	paying (see Table 3, p80)
employee	(Clause P.2)	employing
supplier	(Clause P.3)	supplying (see Table 3, p80)
prohibited	(Clause P.3)	prohibiting (see Table 5, p130)
fulfilled	(Clause L.1)	fulfilling (see Table 11, p143)
violated	(Clause L.2)	violating (see Table 13, p152)
entitled	(Clause L.3)	being entitled
More specifically, 'being entitled' or 'having a right', implies an occurrence of someone else 'being obliged' (see Guideline 14, p111; Table 13, p152).		

Exposing underlying occurrences from role names is an important step in requirements elicitation, as it allows the analyst to identify associations about which further information may need to be recorded. For example, the identification of an occurrence of *employing* points to the need to record not only the holder of the employee role (which is explicit in the specification in our application scenario), but also the holder of the employer role, and start- and end- dates of employment (which are implicit in the specification). Also, *responsibilities* and *privileges* (§4.5) are typically associated with each identified role. During requirements elicitation, an analyst can uncover the norms associated with each role by using templates such as '[role-name] must ...', '[role-name] must not ...', and '[role-name] can ...'. For instance '[applicants] can register for the conference by completing the registration form before the deadline'.

Whilst `Employee` and `Manager` are commonly regarded as classes in object-oriented design (often inheriting from `Person`), an occurrence-centric analysis would recognize occurrences of *employing* and *managing*. A person would then be a current employee for the duration of their participation in an occurrence of *being employed* (by a company), and a person would be a manager through their participation in an occurrence of *managing* (employees).

4.2 Selection Occurrences (Queries)

Modifiers in English are used to select instances based on qualification (matching or conformance with recorded criteria).

Modifiers or qualifiers are often indicated by (explicit or implicit) 'that', 'which', 'where', or 'who' and may imply: a) the existence of a query which describes – that is, is *covering* (§3.3) – a set of individuals, and b) the use of that query for *selecting*, at a certain time, a particular set or sets.	**Guideline 6**

Examples from Application Scenario:

By Guideline 6, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Query in EDEEQL (p84)

employees (that are) older than 25
(Clause P.2)
		`[participants in role [=employee] in [occurrences of [employing]]]`
		`intersection`
		`[participants in role [=aged] in occurrences of [being_aged] where [participants in role [=age] are [>25]]`

payments (that are) more than $10,000 (Clause P.3)		`occurrences of [paying] where [>10,000] is [=paid_amount]` (`query4` in Figure 2, p85)

Various English words or symbols may indicate specific set-operators in EDEEQL (see Appendix 1 for syntax).	**Guideline 7**

Examples:

By Guideline 7, (Provenance)

(Evidence)		(Consequence)
Appearance of Word / Symbol	… indicates …	EDEEQL Set Operator
'and', 'also', 'with', 'that', 'which', 'where', 'who', and adjectives or adjectival clauses e.g. 'low-carbon [adj.] steel'; 'steel that is low in carbon'; 'steel with low-carbon content')		intersection e.g. steel ∩ low-carbon things
'and', 'or', comma (','), semi-colon (';'), bullet (list). e.g. customers and employees		union e.g. customers ∪ employees
'but', 'not', 'except', 'excluding', 'apart from', 'besides', 'without', 'with the exception of', 'save', 'however', 'although' e.g. 'customers but not gold customers'		difference e.g. customers – gold customers

Words used for discourse deixis[10] – such as 'above', 'below', 'earlier', 'afore-mentioned', 'later', 'this', 'here', 'there', 'previous', 'following', 'next', and cross-references to documents or chapter and section headings – may indicate the existence of a query that selects labelled utterances or provisions (see §6.1). Typically, provisions are selected so that they may be voided during conflict resolution (see §6.3), or in order to choose which clauses specify all-things-considered obligations (see §5.6.9).	**Guideline 8**

Examples:

By Guideline 8, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Query
'above'		a query that selects all clauses that appear above the current clause in the current document

[10] 'Discourse deixis' [SIL2002] is a term used in linguistics for expressions that point to other utterances in a verbal or textual discourse.

4.3 Quantification Occurrences

The simplest form of quantification occurrence is *counting*. Occurrences of counting imply the storage and execution, at the appropriate time, of cardinality queries (Section A1.3), as well as, perhaps, the storage of *counting* occurrences that provide a history of such counts. Other forms of quantification include *measuring* (by *observing* or *computing*).

Guideline 9

The English cardinals ('one', 'two', etc.) and the quantifiers 'a' / 'one', 'none' / 'no' / 'not' (and negation affixes such as un–, il–, non–, im–, in–, –less, –free), 'some', 'few', 'multiple', 'many', 'most', 'each' / 'all' / 'every', 'only', 'low', and 'high', may imply counting or measuring, and can also indicate implicit prohibitions or powers (see §4.5). With vague quantifiers, a specific convention – see §4.6 – is typically applied (e.g. 'few, according to clause *x*' is '<3'; 'low, according to clause *y*' is '≤ 10'). The exact convention used should be made explicit to avoid fuzziness in the contract.

Examples:

By Guideline 9, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Occurrences of Type
no		counting, with count = 0
all		counting (where two counts are equal)
e.g. one interpretation of 'all clerks are married' is that '`count (clerks)`' = '`count (clerks participating in role married in occurrences of being_married)`'. Consistent with this is the interpretation of the sentence as 'the participants in an occurrence of being-married are the results of (select the set of clerks)'.		
three management signatures are needed		counting, of occurrences of signing by managers

only managers possess company credit cards	'zero non-managers possess company credit cards' or "`count (occurrences of possessing with credit cards in role [=possessed] and (universe minus managers) in role [=possessor])`; if the result of the counting exceeds zero, the counted items violate policy"

Examples from Application Scenario:

all	(Clause L.1)	counting, where `count of actual occurrences = count of obliged occurrences` (see Table 11, p143)
some	(Clause L.2)	counting, where `count of actual occurrences < count of obliged occurrences` (see Table 13, p152)
low carbon	(Clause D.1)	measuring, of carbon content, and comparing (§4.4) to threshold

'Adverbs of frequency', which stand in the place of the usual quantifiers, may show that occurrences are being quantified over. **Guideline 10**

For instance:

Regular Quantifier … becomes …	Frequency Adverb (Quantifier over Occurrences)
no	never / at no time / not once
all	always
some	sometimes / occasionally
few	rarely / seldom / almost never / hardly ever
many	often / regularly
most	usually / normally / almost always
one	once
two	twice

Examples:

By Guideline 10, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Occurrences of Type
Steelmans never delivered on time		counting, with (count of on-time occurrences of delivering by Steelmans) = 0

The appearance of a numeral, or a unit of measure, in a specification typically denotes that some form of counting or measuring has occurred or must occur. **Guideline 11**

Examples from Application Scenario:

By Guideline 11, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Occurrences of Type
1600 x 400 x 5.0 mm (Clause D.1)		measuring: at least three measuring occurrences, where steel is item_measured and length, width, and thickness are the dimension_measured

Occurrences of measuring – which are comparable in some senses to CANDID's measurement functions [Lee80, p50] – should take a quantity (number), a unit of measure (e.g. 3 metres, 5 managers), an item measured, and a dimension (e.g. height, width). For instance, an occurrence of measuring the thickness of a delivered sheet of steel would be denoted thus:

measuring1 (occurrence instance)

(role)	(participant)
item_measured:	sheet4
quantity_measured:	5.02
unit_of_measure:	mm
dimension_measured:	thickness
measurer:	Bob (Quality Controller 3)

Presuming the sheet also measured 1600mm in length, and 400mm in width, and had a computed carbon content of less than 0.1% (the maximum prescribed by

Euro-Norm 10025 for low-carbon steel) at the time of delivery, this sheet would – at least temporarily – be within the quality specifications of the clause, and would therefore count as being 'steel', in terms of Clause D.1 (see §5.5).

4.4 Sorting and Comparison Occurrences

Comparitives, superlatives, ordinals, ranges, and some function words may imply occurrences of *sorting* (i.e. *ordering* or *sequencing*) or alternatively merely occurrences of *comparing*. Every occurrence of sorting includes occurrences of comparing using some comparator function.

Guideline 12

The existence of implicit occurrences of *sorting* and *comparison* may be indicated variously by:

- ordinals like 'first', 'second', 'third', …, 'last'
- suffixes such as –er, –st, –nd, –rd, and –th (e.g. 'richer', 'wealthiest'), or prefixes like pre–, post–, and suc–
- lexical items such as 'than', 'too', 'exceeds', 'excess', 'enough', 'before', 'after', 'prior', 'different', 'same', 'more', 'most', 'less', 'least', 'worse', and 'worst'.

Examples:

By Guideline 12, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Occurrences of Type
the lowest cost		sorting by cost
the higher priced item		sorting by price
before 1st October		comparing temporal order
too long / long enough		comparing to a deadline or threshold

Examples from Application Scenario:

more than $10,000 (Clause P.3)	comparing to threshold measured in units of dollars
older than 25 (Clause P.2)	comparing to an age threshold measured in years

Ranges are often signalled by means and tolerance levels (e.g. 12pm ± 30 minutes), or by the prepositions and words: 'between', 'in', 'within', 'above', 'below', 'from… to…', 'on', 'during', 'more than', 'less than', 'maximum', 'minimum', 'limit', or 'bounds').	**Guideline 13**

Examples from Application Scenario:

± 0.040 mm (Clause D.1)	comparing to upper and lower bounds

Sorting may be accomplished by storing and executing, at the relevant time, an ordinal query (Section A1.5) or by explicitly tagging items with their predecessor and successor, the means of comparison (comparator function) used, and the time of comparison. Explicitly tagging items can be achieved through occurrences of *preceding* of the form 'x *preceding* y in comparison order z, at time t'.

The nature of the comparator function (i.e. comparison *metric* used) is important, and often needs to be made explicit. For instance, in 'the wealthiest individual' there is clearly some sorting of individuals, but the comparison function used may take into account the cash reserves of the individual, their 'paper wealth' in terms of shares, property valuations, or other criteria. It would not be contradictory for an individual to rate first in a comparison of cash wealth, but to rate lower down in the scale for a comparison of paper wealth done at the same time.

4.5 Normative (Prescriptive) Occurrences

Prescriptive policies define what can or must (or cannot or must not) do what, to what, and when; that is, they prescribe the behaviour of role-players in the system. Obligations (occurrences of *being-obliged*) associated with a named role are the **responsibilities** of the role – e.g. what the user *must* do. Authorisations (permissions and prohibitions) associated with a role are the **privileges** of the role – e.g. what the user *can* do.

Appearances of the modal auxiliaries 'can', 'may', 'shall', 'must', 'has/have to', 'need to', 'should', 'could', 'would', 'ought', 'will', 'has/have the right/authority to', 'is/are entitled to', or negations of these ('cannot', 'may not', …) indicate occurrences of *permitting* (§5.4), *prohibiting* (§5.3), or *being-obliged* (§5.6). Likewise for the suffixes –able and –ible. These indicators may also, or alternatively, indicate the existence of a function which encodes a *power or liability* (§5.5), or *disability or immunity* (§5.3.2).	**Guideline 14**

Examples:

By Guideline 14, (Provenance)

(Evidence) (Consequence)

Appearance of Word … indicates …	*(Prima facie)* Occurrences of Type
acceptable	*permitting* to accept / *power* to bring about an occurrence of *accepting*
tolerable	*permitting* to tolerate / *power* to bring about an occurrence of *tolerating*
payable	*being-obliged* to pay

Examples from Application Scenario:

SkyHi must pay Steelmans (Clause C.1)	*being-obliged* to pay (§5.6)
Clerks may not buy steel (Clause P.1)	*prohibiting* buying (§5.3) *legal disability* of clerks to buy (§5.3.2)
SkyHi has the right to return the steel within 30 days (similarly "The steel is returnable within 30 days") (Clause C.3)	*power* to bring about an occurrence of *returning* in terms of that clause (§5.5)
"steel" shall mean … (Clause D.1)	a *power* to bring about an occurrence of *being-steel* for any item that meets the criteria (§5.5)
party entitled to compensation (Clause C.3)	another party *being-obliged* to do *compensating* (§5.6.2)

Often, 'must', 'shall', 'have to', 'will', or 'ought' indicate the existence of an obligation: 'SkyHi *must* (similarly shall/have-to/will/ought-to) pay Steelmans $25,000' may indicate that 'SkyHi *is obliged* to pay Steelmans $25,000'. Similarly, 'can' or 'may' regularly denote permission: 'SkyHi *can* (may) distribute steel in the East Anglia region' might have the intended reading 'SkyHi *is permitted* to distribute steel in the East Anglia region'. Finally, 'must not' could be read as a prohibition, as in 'Steelmans must not supply to other distributors' which might be read 'Steelmans *is prohibited (forbidden) from* supplying to other distributors'. Caution must, however, be exercised. As explained by Jones and Sergot [JS93], regarding every appearance of the word 'must' or 'shall' as implying an obligation is naïve. Consider that 'Managers must sign purchase orders' may be intended to mean 'Anyone other than a manager is prohibited from signing purchase orders', and does not necessarily imply that managers are obliged to sign purchase orders.

It is also important to realize that in some contexts modal auxiliaries may not refer to legal concepts such as obligation at all. Instead (or in addition), they might indicate occurrences of *predicting*, *expecting*, or *intending* ('It will be delivered

tomorrow'); *requesting* or *suggesting* ('Can you deliver ten tons of steel tomorrow?'); *offering*, *inviting to treat*[11], *volunteering*, or *accepting* ('You can store it in our warehouse'); or *being physically able* ('I can pay you tomorrow'). The distinction between permission and practical ability has been pointed out by Jones and Sergot [JS96]. The difference between predicting and promising has been explored in the literature on speech acts (illocutionary forces) [Aus76, Sea69, SV85] and agent communication languages (ACLs); for instance, in the work on **FLBC** (page 17).

4.6 Conventional (Descriptive) Occurrences

Contracts and specifications commonly encode *legal powers* (§5.5) to bring about certain occurrences. Provisions use particular *names* or *classifications* for items fitting certain criteria. That is, the items are named, called, or classified, *according to a particular clause* as being of that type, only if they conform to certain criteria. One might think of these criteria as somewhat comparable to the fidelity criteria of the speech act literature [Aus76, Tho98]; these are conditions that must be met in order for a naming or christening to be legal in terms of some set of norms. As is evident from Clause D.1, defining 'steel', in our application scenario, a particular word may mean different things – that is, *cover* (§3.3) different observed items – according to different clauses. Consider that an item given the identifier `c1` may colloquially be called 'steel' because it is silver and shiny, but may not be 'steel, according to Clause D.1' because it either does not conform to the requirements of Euro-Norm 10025, or it does not meet the specifications constraining its allowable dimensions in terms of the clause. Steelmans could *physically* call a particular round ball of shiny metal 'steel', but that would not *legally* mean the item is called 'steel, according to Clause

[11] 'Invitation to treat' is a construct of English law intended to capture a non-binding suggestion by a party. Acceptance of a legal offer creates an obligation, whereas acceptance of an invitation to treat has no such result [TB99]. Similarly, accepting a volunteer does not lead to the creation of an obligation under English law, since volunteering implies there is no expectation of payment and, except for the special case of deeds, English law requires the existence of consideration (exchange) for the formation of a valid contract [TB99].

D.1' as the conformance constraints – fidelity criteria – are not met. Subjective interpretations of words or clauses are dealt with in §5.2 and §5.5.

Classification is a means of grouping items that are similar in some respect. In object-oriented techniques, a class is typically defined as a pre-defined, instantiable template to store data about items in a category. An alternative, which we adopt, is to store criteria (queries) separately from the attributes of entities. This makes it possible to reason about classifications: we can determine whether an item is in a certain class (§3.3.2), or ascertain overlaps between the descriptions of classes (§3.3.3). The latter is helpful in finding and resolving conflicts (Chapter 6).

Note that a record of classification criteria used over time (as is provided by the function device we will introduce in §5.5) is useful because names and classifications change over time and we may wish to refer to an entity unambiguously by a new name, or by a previous name in use at some earlier date. Our approach provides a strong conceptual separation between the symbols and the items they represent. It permits different individuals or institutions to assign different names to the same concept. In the case of two or more individuals assigning the same name to different concepts, we can resolve the ambiguity by qualifying the name with the clause (or document or person) assigning the name, or perhaps with the time or context of the naming, in order to select only the intended concept.

Power (§5.5) is encoded in a variety of ways. Nouns (including proper and common nouns), verbs, adjectives and adverbs often have clause-specific meanings, and imply the existence of criteria for items or occurrences in that class. Also, the English conditionals 'if', 'if … then …', and 'when' may indicate that certain occurrences only come about upon the existence of particular conditions, which are described in terms of other occurrences. Similarly, the English prepositions 'to', 'in order to', 'by', and 'through' may indicate that some set of occurrences brings about some other occurrence.	**Guideline 15**

Examples:

By Guideline 15, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Power of Type
The steel is returned if/when it is shipped to the registered address of Steelmans		An act of (successful) *shipping* brings about an occurrence of *returning*
The steel is returned by/through shipping it to the registered address of Steelmans		"
To / In order to return the steel, ship it to the registered address of Steelmans		"

A conventional occurrence of 'returning' is recognized when occurrences fitting a description ('shipping to the registered address of Steelmans') happen. This is one example of a power (here, of SkyHi to return steel) encoded in a contract. Section 5.5 introduces a structured implementation of the notion of *power*.

The English conditionals 'provided', 'providing', 'on condition that', 'as long as', 'so long as', 'unless', and 'in order to', may indicate minimal necessary conditions that must exist in order for a state of affairs to come about. That is, an *immunity* (§5.3.2), or alternatively a *voidance* (§5.6.9), is encoded: unless those conditions exist, the state of affairs is not taken to obtain.	**Guideline 16**

Examples:

By Guideline 16, (Provenance)

(Evidence)		(Consequence)
Appearance of Word	… indicates …	Immunity of Type / Voidance
The steel may be returned provided / as long as / on condition that it is undamaged.		if the steel is damaged, then (according to this clause), `count (returnings of this steel) = 0`, or, alternatively, all obligations resulting from this return are *voided*.
The steel may be returned unless it is damaged.		"
In order to return the steel, it must be undamaged.		"

Caution should, as always, be used in applying these guidelines. For instance, 'in order to' may indicate intention, expectation, or recommendation (occurrences of *intending*, *expecting*, or *recommending*), rather than existence of legal power or a guarantee. For example, in …

- The steel may be trucked by CargoCarriers in order to return it.

… the specification certainly seems to indicate a permission (based on the appearance of 'may' as described in §4.5). However, it is not clear whether it is meant to confer a *power* to return steel via that means, or merely gives a *recommendation* as to how steel could be returned. Less forcefully even, it might simply give an *expectation* as to how the goal might be achieved. That is, it might just say that occurrences of trucking steel back with CargoCarriers are intended or expected to result in occurrences of returning the steel. Disambiguation of these various senses is

important in contracts, as a party may wish to recommend a course of action to another party, but may not wish to confer a legal power to the other party. Specifying explicitly that this is a recommendation or expectation, rather than a conferral of power or guarantee, may reduce misunderstanding and avoid contractual disputes.

4.7 Summary

☑

Requirement 20 (pg 41): Though it is perhaps not fully machine-automatable, the progression from English specifications to machine interpretable policies should be further systematized.

This chapter has introduced and documented a spectrum of techniques for exposing occurrences in an English language specification. Though we have attempted to be thorough, the rules we have set out are by no means exhaustive, and we have also not considered counterexamples. The intention was simply to provide helpful insights that may be employed by an analyst when uncovering logical structures in natural language specifications. We have shown how to move from varying expressions of semantics in English to canonical forms – for example, various ways of expressing negation in English, such as 'no' and 'un–', can be rendered as the canonical form '*counting*, with count=0' (§4.3); and diverse prescriptive expressions like 'must', 'have to', and '–able' might be rendered as implying occurrences of *being-obliged* (§4.5).

We qualified our proposals with the reminder that, with the current state-of-the-art in natural language understanding technology, the process cannot hope to be mechanistically systematic. Each of the indicative keywords and fragments we have mentioned in this section point to the existence of instances of certain types of occurrences. We seek here only to capture a subset of meanings that may be useful in enlightening the legal and semantic structure of an English language specification of contracts, policies, or regulations. Our guidelines are useful in that they make explicit some helpful rules that analysts can employ when developing structured representations of natural language specifications. The intention is to move some

way towards codifying the hitherto unsystematic transition from analysis to implementation (§2.3.3).

In the next chapters, we explore in detail the types of provisions in a specification, and investigate mechanisms for storing, enforcing, and consistency-checking these provisions.

Chapter 5

Representing Provisions

Kimbrough's Disquotation Theory (op cit, page 17) – a formal theory about sentences that embed propositional content – expounds the fulfilment and violation conditions for speech acts such as asserting, permitting, and obliging. In Chapter 3 we examined the notion of an occurrence and provided a structural representation of this abstraction. This chapter, and the chapters that follow, aim to show how an extended version of Disquotation Theory can be profitably applied to the creation of computational environments for monitoring and enforcing electronic commerce contracts using pervasive, mainstream industrial technologies such as Java and relational databases. In this chapter, we show how contractual provisions – obligations, permissions, prohibitions, and powers – can be represented and stored. In Chapter 6, detailed examples illustrate how a query coverage-determination mechanism can be used to check inter-organizational contractual provisions against internal policies and external legislation for dynamic conflicts. A conflict resolution mechanism that takes into account the life-cycle of individual instantiated obligations is proposed. Finally, in Chapter 7 we look at approaches to monitoring occurrences and enforcing provisions. The work presented in these chapters demonstrates that our extended version of Kimbrough's theory presents a new and promising means of storing interrogable and executable specifications for e-commerce workflow applications.

☑
Requirement 8 (pg 29):
A large and growing number of machine-enforceable rules should be controlled and managed in a database, not haphazardly distributed in text files.

We begin this chapter with a contextual overview of our approach to storing and executing provisions (Section 5.1). In Section 5.1.1, we give a brief review of Kimbrough's Disquotation Theory. Section 5.1.2 explains how Kimbrough's notions may be implemented in software: we show that stored queries can be used as the implementation mechanism of Kimbrough's quotation construct that is used for bracketing propositions within deontic statements. Storage schemata for assertions and for common provisions such as prohibitions, permissions, powers and liabilities, and obligations are shown (Sections 5.2 – 5.6), and we demonstrate how functions are used to determine the necessary deontic and legal consequences of particular provisions.

5.1 Context

Implicit and explicit contracts are an important mechanism for co-ordinating organizational activities. In their contracts, organizations express which states of affairs are desirable and undesirable. They specify which legal states of affairs are achievable, and how, and which are not. In each case, a subjective attitude towards propositional content is given: the state of affairs is obliged, forbidden, permitted, obtains in law, or does not, according to some clause. We demonstrate here that representing and reasoning about subjective, propositional content has applications in the automation of commerce.

In our novel software realization, we adopt a declarative, event-driven approach, where each contractual provision is explicitly stored and monitored, and both software and human components may consult the provisions to determine what currently holds, and what to do next.

We view a **contract**, **policy**, or **regulation**, as a set of provisions. A **provision** specifies an obligation (§5.6), permission (§5.4), prohibition (§5.3), or power (§5.5). Provisions are described in clauses. Broadly, provisions represent rights and duties. We wish to represent and store these provisions and their contents as records in a database.

☑

Requirement 29 (pg 60):
Provisions, whether emanating from contracts, policies, or laws (inter-, intra-, or extra-organizational provisions), should be uniformly represented, to facilitate consistency checking.

Figure 8 presents the general architecture of our EDEE environment. An active database wrapper compares inbound occurrences against stored contract terms, firing consequent obligations and permissions, and bringing about conventionally accepted states of affairs. As illustrated in Figure 8, the wrapper accepts new occurrences, checks which descriptions they fit (1) and which contractual provisions therefore apply (2). The wrapper then infers obligations (3), which can be asynchronously fulfilled by a fulfilment engine (4). The fulfilment occurrences can then be added to the occurrence store where they may bring about obligation satisfaction.

The database is not intended as a general e-marketplace, but rather as a repository of the current contracts and policies of a single company. Together these contracts and policies represent the workflow system specification for that company. Our system is intended for inter-organizational contracts, internal policies within an organization, and external legislation, since all specify the norms that govern the organization and guide execution. Execution components use the database to ascertain what they are obliged to do: that is, what procedures to enact. Users may query the database to ascertain what legal relations have obtained as a result of past occurrences.

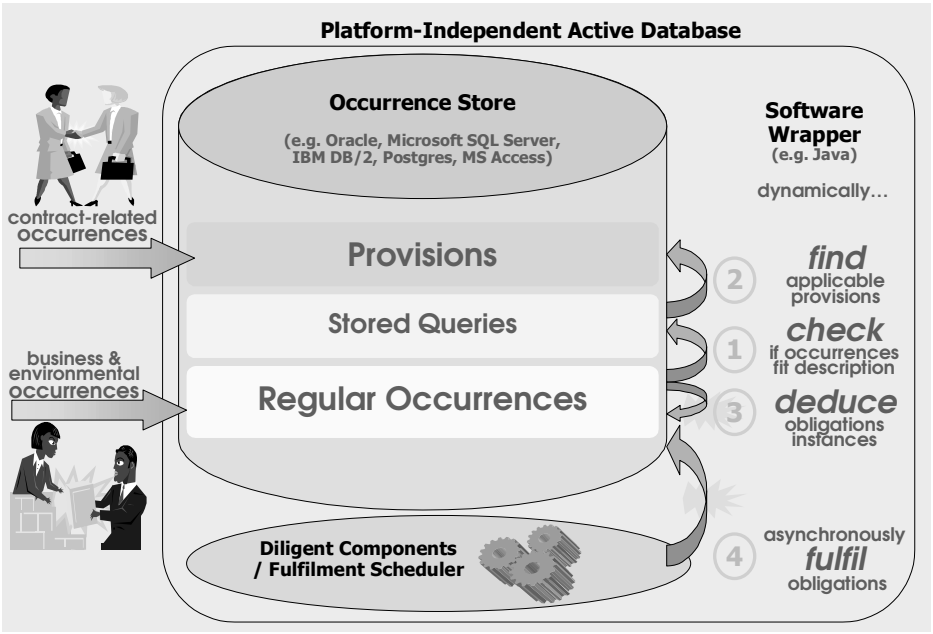


Figure 8: General architecture of EDEE

Existing event-, rule-, and policy-based approaches (§2.1, §2.2, §2.3) assume that components lack free will and execute all their obligations immediately and without delay (§2.2.5). In commercial environments though, agents typically have more flexibility in scheduling the fulfilment of their obligations. Immediate enactment is one possible option, which may or may not minimize overall response time or maximize profitability. With commercial obligations, it is often the case that agents can fulfil an obligation by acting at any time before a deadline or within an interval. Furthermore, while we agree that in the ideal world all obligations are fulfilled, this ideal is unattainable because of real world practicalities such as resource limitations, conflicting obligations, unpredictable environmental forces, and free agents; thus there is a possibility of deviation from prescribed behaviour [DDM2001]. The intention of obligations is not only to specify what should ideally take place, but also what happens when the ideal cannot be met. We must qualify our assumption of free will though, since we do not mean to imply that the software components are simply free to ignore obligations. Rather, our architecture requires that software (and human) components are *diligent*: they consult the database to ascertain what

obligations currently apply, or are notified of this by the active database, and attempt to fulfil them within their resource constraints. When they fail, or indeed when they succeed, they are forced to accept the obligations construed by the database, and the occurrences that are deemed to have occurred as a result of their action or inaction.

Unlike Standard Deontic Logic (page 63), which assumes that being-obliged to do something which is prohibited is a logical contradiction (obligation and prohibition are treated as inter-defined *operators*), we assume that prohibitions and obligations are independent *entities* (i.e. variables that are quantified over), and conflicting norms can exist. An agent may find itself violating a prohibition in order to fulfil an obligation, or vice versa. Entirely avoiding such conflicts is not our intention since we believe that in a world of multiple norm-givers, conflicts and consequent trade-offs are inevitable. Indeed, even a single norm promulgated by a party can result in conflict. Consider Hansson and Makinson's example (op cit, page 65) of a single imperative 'the doctor must immediately visit heart attack victims' producing conflicts if multiple patients, living at remote locations in the rural outback, suffer heart-attacks in quick succession. Hansson and Makinson's view is that, since conflicts cannot exist, one of these norms must be blocked (restrained from the output set of norms) to eliminate the conflict. Our view is that it is undeniable, prima facie, that both obligations exist. It may be that one is violated if the doctor visits only one of the patients. More likely though, one of the obligations is voided by a separate principle of fairness which effectively specifies that a violation in this case is forgivable. Other principles may specify another obligation, to visit the second patient within a more lenient time interval, that can be fulfilled by the doctor. Some ordering criteria must specify which patient is 'second'.

We now turn to a review of Kimbrough's Disquotation Theory (op cit, page 17) and the extensions, alterations, and implementation mechanisms we propose to facilitate executable specification of e-commerce applications.

5.1.1 Kimbrough's Disquotation Theory

In this section, we illustrate briefly Kimbrough's representation of a number of sentence types with embedded propositions: assertions, permissions, prohibitions, and obligations.

Assertions: An instantiation of 'Mary asserts that SkyHi paid $25,000 to Steelmans' from Kimbrough's axiom schema for assertions would yield:

$$\exists e_1 \, (\, asserting(e_1) \wedge Subject(e_1,Mary) \wedge Object(e_1,[\phi]) \,) \rightarrow (Veridical(e_1) \leftrightarrow \phi)$$

Formula 2

where ϕ represents the predicates from *Formula 1* (page 81) which say, in brief, 'Steelmans was paid'. The brackets '*[...]*' are special quotation operators that turn their contents into an opaque string; thereby preventing 'Mary asserting that Steelmans was paid' from necessarily entailing 'Steelmans was paid'. This says that Mary's assertion that SkyHi paid $25,000 to Steelmans is veridical (true) if and only if there was an actual occurrence of SkyHi paying $25,000 to Steelmans.

Prohibitions: In Kimbrough's semantics, a prohibition against SkyHi paying more than $10,000 could be expressed as:

$$\exists e_2 \, (\, prohibiting(e_2) \wedge IsAccordingToClause(e_2,c) \wedge Object(e_2,[\phi]) \,) \rightarrow (Violated(e_2,c) \leftrightarrow \phi)$$

Formula 3

where ϕ is an expression describing an occurrence of SkyHi paying more than $10,000. Thus we say that the prohibition against SkyHi paying Steelmans is violated, in terms of clause *c*, if and only if, more than $10,000 is actually paid to Steelmans. It should be noted that we have, here and elsewhere, supplanted Kimbrough's original suggestion of an *InSystemOfNorms(e,n)* predicate, with an *IsAccordingToClause(e,c)* predicate, to capture the more specific sentence '… is violated, according to this *clause* (utterance) *c*, if …'. We believe this adjustment is necessary as systems of norms (sets of regulations) may contain conflicting provisions that can only be resolved through choice between identified clauses. Therefore, in each provision it is only appropriate to authoritatively state that 'there is

violation *in terms of this specific clause*', but not 'there is violation in terms of this system of norms', since other provisions (clauses) in the system of norms may conflict and override. For instance, the so-called *de minimis* provision of English Law [TB99, p144] may rule that there is no violation in the event that SkyHi paid only $24,499, since the difference is not material.

Permissions: In Kimbrough's semantics, the permission of SkyHi to pay Steelmans could be expressed as:

$$\exists e_3 \, (\, permitting(e_3) \wedge IsAccordingToClause(e_3,c) \wedge Object(e_3,[\phi]) \,) \rightarrow (\neg Violated(e_3,c))$$

Formula 4

Kimbrough reads this as 'permissions cannot be violated'. More particularly, we would read this as 'no violations are brought about by permitted occurrences'. A similar interpretation of permission, following Anderson [And58], is provided in [Lee88], who says that a state of affairs is permitted if and only if it will never be the case that bringing about that state of affairs implies sanctions. We prefer the Kimbrian interpretation, since we see violations as being distinct from sanctions, as the former does not necessarily imply *redress* (penalty), merely *transgress*. Lee's interpretation receives support from Cholvy, Cuppens, and Saurel [CCS97], who argue that it is prejudicial that p is the case if and only if it is necessary that if p occurs then it is obligatory to repair damage. However, we wish to emphasize that the fact that something is wrong (there is a violation or offence) is distinct from any penalties – 'sanctions' [Lee88] or 'reparations' [CCS97] – that follow from that violation. That someone has been wronged does not automatically imply an obligation to repair the wrong: there are many cases where no reparations are specified or available, yet the violation is still deemed to have occurred. Many contractual obligations do not actually have specific sanctions explicitly noted both because parties may prefer to leave things open for future resolution should the need arise, or because, at least under English Law, sanctions are not automatically imposed unless the injured party initiates litigation and is successful. Thus, violations may exist without sanctions.

Another useful sense of the term 'permission', commonly attributed to Bentham [Lin77], is permission in the sense of 'vested liberty'. A vested liberty obliges others not to prevent the action [Lin77, p17], whereas a naked liberty allows other people to prevent the action in question. Violation (through attempting to prevent a permitted occurrence, or more specifically, through attempting to interfere with a vested liberty) can bring about a set of obligations on a liable party.

Obligations: In Kimbrough's semantics, the obligation of SkyHi to pay Steelmans could be expressed as:

$$\exists e_4 \, (\, ought(e_4) \wedge IsAccordingToClause(e_4,c) \wedge Object(e_4,[\phi]) \,) \rightarrow (Violated(e_4,c) \leftrightarrow \neg\phi)$$

Formula 5

This says that the obligation of SkyHi to pay Steelmans is violated, in terms of clause *c*, if and only if, SkyHi did not actually pay $25,000 to Steelmans. Kimbrough here uses a variant of Anderson's [And58] reduction, which says that 'ϕ ought to be the case' unpacks to 'necessarily, if ϕ is not the case, violation happens'.

5.1.2 An implementation of Kimbrough's Disquotation Theory

Kimbrough's logic is, by and large, appealing; certain alterations were proposed above (§3.1, §5.1.1). What we desire is a database representation that mimics this logic. We suggest an implementation in Java which models the quoted contents $[\phi]$ as a query – specifically, a query that returns occurrences. The quoted contents cannot be modelled as facts as this would imply that the occurrences actually occurred, when in fact they were only *described*. That is, the occurrences enclosed in a disquotation operator should not be regarded as actual occurrences, but *descriptions* of occurrences. For instance, if obliged occurrences were regarded as actual (rather than described) this would imply that all obligations are kept, when indeed obligations may be violated should the described occurrences never happen. Thus it is proper to model only the criteria that describe the occurrences, since these describe what nature of occurrence was obliged, without the undesirable implication that the described occurrences were actually realized. We provide a continuous query mechanism,

which determines incrementally, as data is added to the database, which stored queries produce new results or consume existing results. This mechanism enables the system to determine whether an occurrence is obliged, permitted, or prohibited, by determining whether the occurrence fits a stored query.

☑
Requirement 2 (pg 20):
Occurrences with nested propositional content must be expressible.

When we store obligations, permissions, prohibitions, and powers, what we are putting into the database are representations of expressions with propositional content: e.g. x permits that *P*, it is obligatory that *Q*, y prohibits that *R*, *S* brings about *T*. We know that the *expression* occurred since we can be sure the obligation, permission, prohibition, or power was once given or *expressed* (even if it has since been overridden). However, we do not assume that the *disquoted propositional content* – *P*, *Q*, *R*, *S*, or *T* – is true, since permissions are not always used, obligations are not always met, prohibitions are not always violated, and powers are not always exercised. We must then use a query – a descriptive set of criteria – to store the disquoted propositional content, rather than use an actual occurrence. This captures the intuition that the disquoted content was *described*, but may not have actually *occurred*. The benefit of using stored queries – that is, stored descriptions – is that, when and if the content (occurrence or set of occurrences) does happen, we can detect that the occurrences (eventualities) are covered by a stored query. A continuous query mechanism, or 'coverage-checker', is employed for this purpose. Upon insertion of a new occurrence, the coverage-checker determines which stored queries cover the new occurrence, and which expressed provisions pertain to the occurrence.

The remainder of this section revisits examples of assertions, prohibitions, permissions, and obligations from our application scenario (§1.2), demonstrating implementations of Kimbrough's constructs. We also add notions of powers, liabilities, disabilities, and immunities, inspired by Hohfeld [Hoh78], which we have found necessary.

5.2 Assertions

The case of assertions is unique, in that we have decided not to adopt the disquotation operator for reasons of the inherent subjectivism of our approach. By this we mean that we have chosen a world-view where all recorded 'facts' are taken as subjective opinions. For our purposes, we do not find it meaningful to determine whether an assertion is 'veridical' since we are unable to compare to an objective truth: only subjective opinions – *X* is the case according to this clause, and *Y* is the case according to another clause – are stored.

For the anomalous case of assertions then, our approach has been to supplant the disquotation operator with an alternative: we append the conjunct *IsAccordingTo(e, ClauseX)* to all occurrence descriptions. This captures the notion that all occurrence descriptions are prima-facie construals relative to an utterance (identified by a system-defined clause identifier). Our database representation assumes that, while objective truth exists, it may sometimes be arguable as to what it is – all occurrence descriptions are relative to some clause. For instance, 'The clerk, John, bought steel, according to Clause 8' does not mean that the steel was certainly bought (for other regulations may override); only that Clause 8 states this. That 'John bought steel' is therefore the opinion of Clause 8, and is merely prima facie, not conclusive, evidence of buying. Our rules must be specific. We must differentiate 'buying according to Clause 8' from 'buying according to Clause P.1', since it is possible, in the case where clerks are not empowered to buy steel and John is a clerk, for there to be a buying in terms of the former clause, but not in terms of the latter. In contrast to many traditional views of data representation, in our database we record what has been said, rather than attempting to capture what 'is': when we ask the database what is, we are more specifically asking what is according to a certain utterance (clause).

It is worthwhile to note here that many approaches to evaluating subjective evidence are available. In our *qualitative* approach, occurrences according to a particular clause are brought about, through a function, whenever there are occurrences in a particular domain – that is, whenever an occurrence fitting a

convention is recognized. An interesting and complementary *quantitative* approach to assessing the believability of subjective opinions of various agents in a distributed setting is provided in Dimitrakos and Bicarregui [DB2001] and Daskalopulu, Dimitrakos, and Maibaum [DDM2001]. A believability metric, computed by numeric combination of weighted opinions, is associated with each proposition. Roughly, Dimitrakos and co-authors have it that 'pizza A is delivered' if Susan believes with 0.8 probability that it was delivered, Peter believes with 0.2 probability that it was delivered, and Susan's opinion is weighted more strongly than Peter's, based on their reputations and credibility in the area. In contrast, we might have it that 'pizza A is delivered, according to Clause 8.6 of the delivery contract between Peter and Susan' whenever Peter says it has been delivered, irrespective of Peter's credibility. Such a construal of what 'delivery' means is common in 'customer satisfaction guaranteed' contracts. Clause 9.1 of the Uniform Commercial Code may specify different criteria as to what it means for a good to have been delivered, and we may need to choose, in accordance with explicit legal principles, which construal overrides in the circumstance. Our approach to subjectivity of information is one of 'locality of inference', whereas Dimitrakos and co-authors deal with the aspect of 'reliability of evidence'.

5.3 Prohibitions

We distinguish here between two types of prohibitions: violable prohibitions, and inviolable prohibitions (legal disabilities).

5.3.1 Violable prohibitions

Violable prohibitions admit the possibility of violation. This is the sense of prohibition used by Kimbrough. If an occurrence fits the description of occurrences prohibited by a particular clause, it can be said to be prohibited in terms of that clause, and its existence brings about a violation of that clause. Consider Clause P.3 ('payments of more than $10,000 to suppliers are prohibited') from our

application scenario of §1.2. The 'violable prohibition' sense of this clause can be modelled as shown in Table 5. Here, Query10 is a pointer to a query describing the set of prohibited occurrences: in our example, occurrences of a supplier being paid more than $10,000. The representation of Query10 was shown on page 84.

Occurrence	Role	Participant
prohibiting1	prohibited	Query10
	isAccordingTo	Clause P.3
violating_function1	domain	Query10
	isAccordingTo	Clause 82
	violated	prohibiting1

Query10 (see page 84) = occurrences of paying where more than $10,000 is in role=paid ∩ occurrences of paying where a supplier is in role=payee

Table 5: A schema for storing a violable general prohibition

violating_function1 is an identified function, whose domain is Query10 and whose range is a set of occurrences violating_function1(Query10). We append the occurrence type produced by the function to the start of the function name to make the output range clearer and more easily accessible to the query mechanism. Therefore, an occurrence, paying1, as defined in Table 3, which is covered by Query10 and hence in the domain of violating_function1, would produce an occurrence which we might refer to as violating_function1(paying1), which can be seen (Table 5) to be an occurrence of *violating*, where it is the general prohibition, prohibiting1, that is violated. The shorthand identifier violating_function1(paying1) may be thought of as referring to the following specific occurrence that could be explicitly stored in the occurrence store:

violating1 =		(legal consequence)
source_rule:	violating_function1	(provenance)
source_occurrence:	paying1	(evidence)
violated:	prohibiting1	

violating1 (where 1 is a unique identifier) captures the particular legal consequence brought about from applying rules of exact provenance (violating_function1) to specific evidence (paying1). The link between a rule (function), a happening, and the resultant

conclusion is recorded by storing the sources of the conclusion as its source_rule and source_occurrence attributes.

It should be noted that the occurrence, prohibiting1, is a *general* prohibition, which generates *specific* prohibition *instances* for each prohibited occurrence. As shown in Table 6, the function device is used to produce case-specific instances of prohibitions, from the general prohibitions.

Occurrence	Role	Participant
prohibiting_function1	domain	Query10
	prohibited	|Query10|[12]
	isAccordingTo	Clause P.3

Query10 (see page 84) = occurrences of paying where more than $10,000 is in role=paid ∩ occurrences of paying where a supplier is in role=payee

Table 6: Generation of specific prohibition instances from general prohibitions

Assuming that paying1, a payment of $25,000 to Steelmans (Table 3, page 80), occurred, we could deduce the existence of the following specific prohibition instance:

prohibiting2 =		(legal consequence)
source_rule:	prohibiting_function1	(provenance)
source_occurrence:	paying1	(evidence)
prohibited:	paying1	

prohibiting2 (another name for the occurrence also identifiable as prohibiting_function1(paying1)) may be interpreted as saying that there is, *prima facie*, a specific prohibition against the payment, paying1.

[12] The notation |…| allows us to refer to the originating occurrence that brought about the state of affairs produced by the function. We can refer to the attributes of the originating occurrence by embedding |…| into query expressions from our language. To give an example from Table 6: assuming the occurrence paying1 is in the domain denoted by Query10, it then brings about an occurrence prohibiting_function1(paying1), alias prohibiting2, where paying1 is substituted for |Query10| in the prohibited role of prohibiting2.

The function construct is comparable in some senses to Jones and Sergot's [JS96] counts-as connective, $\Rightarrow_S$, and Goldman's conventional-generation connective, though our functions are relativized to an identified clause (utterance), C, rather than relativized to an institution, S, as for the counts-as connective, $\Rightarrow_S$. Thus, it is the clause that deems that a certain occurrence is brought about, rather than an institution. This is necessary because clauses are atomic, whereas current institutional interpretations can only be selected once all *prima facie* atomic clausal interpretations have been compared. Further, Jones and Sergot's connective, $\Rightarrow_S$, is used in the context of Hohfeld's [Hoh78] notion of power, where an agent is only empowered to bring about ideal states, whereas we follow Goldman's construal that a non-ideal state, such as an occurrence of violating, may be generated by conventional occurrences (occurrences in the domain of an identified function).

Again, we point out that all provisions in a contract should be regarded as subjective construals. 'The prohibition is violated' should *not* be taken to mean that the prohibition is authoritatively violated; but should rather be taken as a partial rendering of the sentence 'The prohibition is violated *according to some clause*'. Another clause may provide that the prohibition is not violated, and a choice of which clause to accept as authoritative is then required. The conflicting clauses may even originate from the same institution. Determining the overarching construal of the institution involves choosing which clause applies under the circumstances.

5.3.2 Inviolable prohibitions (disabilities / immunities)

Inviolable prohibitions are often termed legal 'disabilities' or 'immunities'. Inviolable prohibitions cannot be violated, since the prohibited party simply doesn't have the legal power to bring about the prohibited state. Under Hohfeld's [Hoh78] terminology this would be called *disability*. We could take Clause P.1 of our application scenario (§1.2) as specifying that clerks have no authority to purchase steel. Table 7 illustrates the representation of this sense of the clause: it specifies that the number of purchases of steel by a clerk under this clause is zero (the count of

results produced by Query501 is 0). This means that no action by them can bring about a purchase in terms of this clause.

Occurrence	Role	Participant
counting1	counted	Query501
	count	0
	isAccordingTo	Clause P.1

Query501 = occurrences where a clerk is buyer ∩ occurrences where steel is bought

Table 7: A schema for storing a disability or immunity (inviolable prohibition)

A similar construction can be used to implement Hohfeld's [Hoh78] notion of *immunity*: under immunity, no action by any party can bring about a certain state of affairs (e.g. no party can bring about an occurrence of a particular party being obliged to do something). The party that benefits from this immunity is said to be immune. In Table 7, the description (Query501) does not specify the seller; the implication is that steel purchases by a clerk from any seller may not come about. This means that, according to this clause, everyone has immunity from entering into steel purchases with a clerk.

☑

Requirement 25 (pg 45): Physical occurrences must be distinguished from legal occurrences. Both must be recorded.

Finally, note that when a policy states 'clerks may not purchase steel', this may be intended as implying the existence of *either* a violable prohibition, an inviolable prohibition, or *both*. We do not intend the choice to be an either/or choice between the types of prohibition, since often the intention is that there is both a violable and an inviolable prohibition in existence. English specifications of policy are problematic since it is not clear whether 'may not' is intended to imply the existence of a violable prohibition, *or* an inviolable prohibition (legal disability or immunity), or *both*. Our purpose in distinguishing between violable and inviolable prohibitions is so that we may disambiguate English specifications of policy and be more particular as to which type of prohibition is present. In this case, assume we mean for there to be both a

violable prohibition and an inviolable prohibition (legal disability or immunity). If the clerk indeed proceeds to make a purchase, he violates the violable prohibition *and* his purchase is still covered by the inviolable prohibition which prevents it from having effect. It is tempting to say that the clerk exercised his power in violation of the prohibition, but this is misleading. In fact, the clerk did not exercise his power in this case, since he performed a purchase, but not a 'purchase in terms of Clause P.1' so no legal power was really exercised in that respect (because the clerk in fact had a legal disability). He is still guilty of a violation, and perhaps subject to sanctions against him, since it was purchases of any sort that were forbidden, and even though he did not perform a 'purchase in terms of Clause P.1', he did perform a purchase according to some other clause. The fact that the purchase is not effectual in terms of Clause P.1 does not mean that it doesn't violate a prohibition against all purchases, since it is still nonetheless a purchase in some other sense.

5.4 Permissions

'Employees older than 25 may buy steel (Clause P.2)', from our application scenario (§1.2), is also an ambiguous expression. It may refer to employees older than 25 being *permitted* (allowed) to buy steel, or to employees older than 25 being *empowered* (that is, legally capable in terms of a clause) to buy. As pointed out by Jones and Sergot [JS96], English expressions such as '… having *authority* to…' are vague between (at least) these two senses of 'having permission' versus 'having power'. A similar vagueness arises in the English notion of 'exercising a right' which may mean 'doing something that was permitted' or 'bringing about a state of affairs through use of a conferred power'. Permission is dealt with here, whilst we leave empowerment for the next subsection (§5.5).

We distinguish between two types of permissions: violable and inviolable permissions.

5.4.1 Violable permissions (vested liberties)

Violable permissions are the Benthamite 'vested liberties' referred to by various authors [Lin77, Mak86, NSJ98], and discussed in §5.1.1. This sense of permission can be represented as a violable prohibition (§5.3), where the prohibited occurrences are described as 'any occurrences of *attempting* to prevent, or actually *preventing*, the permitted occurrences'.

Related to the notion of vested liberties is the legal principal that protects the right of the beneficiary to receive the intended benefit from the obligation by prohibiting actual or attempted interference. For instance, an obligation to mail a letter generally has implicit within it a prohibition against interfering with the letter being received. The receipt of the letter is the state of affairs that is the intended effect (benefit) of the obligation. Thus, burning the mailbox that contains the letter would be implicitly prohibited as it interferes with the intended beneficiary's right to receipt. In Abrahams and Kimbrough [AK2002] we show our treatment in event semantics of what we term "arson's reduction" or the problem of vacuuming conjunction. Here the co-existence of an actual interference, or even intention or attempt to interfere, or simply expectation of action futility, violates a prohibition-against-interfering-with-intended-effects that is implicitly associated with every obligation.

5.4.2 Inviolable permissions (privileges)

Inviolable permissions are the sense of permission used by Kimbrough. Inviolable permissions are not directly comparable to Benthamite 'naked liberties' since the concept of naked liberty entails only that interference by other parties is not prohibited, but does not explicitly specify that performing the permitted action can never lead to a violation as Kimbrough [Kim2001] and Lee [Lee88], following Anderson [And58], suggest. That is, by 'inviolable permission' we specifically mean to capture the notion that performing any action covered by the permission does not lead to any violation. This subtlety is not captured by the term 'naked liberty'. Consider an example. '*Employees older than 25* are permitted *to buy steel*' embeds the

query 'select purchases by employees older than 25 of steel'. That is, according to this clause, any occurrence fitting this query is permitted and does not bring about any violation. The permission implies the count of violations brought about by occurrences of this nature is zero. Table 8 illustrates. Inviolable permissions are comparable in some sense to Hohfeldian *privileges*; privileges [Hoh78] imply no duty to refrain exists – i.e. no violation is brought about by indulging in the action.

Occurrence	Role	Participant
permitting1	permitted	Query502
	isAccordingTo	Clause P.2
counting1	counted	Query503
	count	0
	isAccordingTo	Clause 2.6

Query502 = occurrences where employee older than 25 is buyer ∩ occurrences where steel is bought

Query503 = occurrences of violating brought about by occurrences in Query502

(i.e. occurrences of violating where source_occurrence is in Query502)

Table 8: A schema for storing an inviolable general permission (privilege)

It should be noted that the occurrence, permitting1, is a *general* permission, and, as shown in Table 9 below, a function can be defined to generate *specific* permission *instances* for each permitted occurrence.

Occurrence	Role	Participant
permitting_function1	domain	Query502
	permitted	|Query502|
	isAccordingTo	Clause P.2

Table 9: Generation of specific permission instances from general permissions

Finally, as we pointed out for prohibitions above, we do not mean for the terms 'violable' and 'inviolable' to imply that the two types of permissions cannot co-exist

in relation to a particular set of described occurrences. Nor do we mean that a particular English sentence implies one or the other. Indeed, violable and inviolable permissions often do co-exist, and a given English sentence may ambiguously imply either or both. Consider 'Employees older than 25 may buy steel (Clause P.2)'. This may be taken to imply *either* or *both* of:

- Anyone who interferes with such an employee's right to buy steel is in violation (a violable permission), or,
- Such an employee buying steel does not bring about a violation in terms of this clause (an inviolable permission). Should another clause prohibit an individual who is such an employee from buying steel, we can see that this prohibition conflicts with the permission, since the prohibition implies that violations are brought about by such purchases, whereas the permission implies that no (zero) such violations are brought about.

5.5 Powers and Liabilities

In our variation of Jones and Sergot's usage [JS96]: powers encode the ability of an actor to bring about a state-of-affairs according to a clause. Consider the rule 'Clause P.2: Employees older than 25 may buy steel'. The reading of this as a power of the employee may be represented as shown in Table 10.

☑

Requirement 35 (pg 70): Primitives governing change of legal relations – such as power and immunity [Hoh78] – must be provided.

Occurrence	Role	Participant
buying_function1	domain	Query504
	isAccordingTo	Clause P.2
	buyer	Query505
	bought	Query506
returning_function1	domain	Query507
	isAccordingTo	Clause C.3
	returned	Query508
	returner	Query509

Query504 = (occurrences where an employee older than 25 is buyer ∩ occurrences where steel is bought) – occurrences where Clause P.2 is isAccordingTo[13]

Query505 = participants in role buyer in occurrence |Query504|

Query506 = participants in role bought in occurrence |Query504|

Query507 = (occurrences of returning where count of days since (occurrence of purchasing for participant in role returned) is ≤ 30) – occurrences where Clause C.3 is isAccordingTo

Query508 = participants in role returner in occurrence |Query507|

Query509 = participants in role returned in occurrence |Query507|

Table 10: A schema for storing powers

In Table 10 the function buying_function1 encodes the power to bring about a purchase. Clearly, the only means of bringing about a *purchase in terms of Clause P.2* is to insert an occurrence in the domain (Query504) of the function. Assume buying4 is a purchase of steel by the 40-year old manager Marge. buying4 is covered by Query504 and therefore is in the domain of the function buying_function1(). Substituting buying4 for Query504 in buying_function1(Query504), the function therefore produces the occurrence instance identified as buying_function1(buying1). It should be noted that, as the database wrapper automatically tags each occurrence with a clause identifier upon addition to the database, simply adding an occurrence of buying to the database does not make it a *purchase in terms of Clause P.2*. So a purchase, say buying6, of gold by an underage dispatch clerk would not be construed as a *purchase in terms of Clause P.2*, since buying6 is not covered by Query504 and therefore is not in the domain of buying_function1.

[13] This third set criterion is necessary in order to prevent infinite recursion: that is, to prevent purchases according to Clause P.2 from bringing about purchases according to Clause P.2. The symbol '-' can be read '… but not …'.

Similarly, in Table 10 the function returning_function1 encodes the power to bring about a valid return, in terms of Clause C.3, through a physical return within 30 days of the purchase occurrence relating to the return.

Definitions of terms – that is, forced construals – are implemented in a like manner: we could define functions, with appropriate domains, that bring about occurrences of being_steel (to implement Clause D.1). As suggested by the appearance of numbers and units of measure (see analysis guidelines in §4.3), the domain of the being_steel function would need to refer, inter alia, to occurrences of measuring. An occurrence of being_steel in terms of Clause D.1 then comes about only when `all`[14] of the following occur: there is an occurrence of the item being_Fe360 and this isAccordingTo Euro-Norm 10025; and there is a measuring where that item is item_measured and quantity_measured is 1600, unit_of_measure is mm, and dimension_measured is height; and there is a measuring of that item where quantity_measured is 400, unit_of_measure is mm, and dimension_measured is width; etc.

Notice that the mechanism of using functions to bring about states-of-affairs-according-to-a-clause implements not just *powers*, but also *liabilities*, which, according to Hohfeld [Hoh78], are the correlative of powers. Hohfeld does not here use 'liability' in the accounting sense of debt or "having an obligation towards another party"; rather, he makes use of 'liability' in the sense of "being subject to having one's legal relationships altered by another party". As Hohfeld explains, his sense of liability does not necessarily imply disadvantage to the liable party; liability implies merely that the party is subject to having their legal relations altered by virtue of an occurrence. The notion of power implies that an actor is empowered to bring about a state of affairs[15] through their own action; what Hohfeld terms 'volitional control'. The notion of liability is similar, though it implies that the state of affairs may be brought about by another party, by an environmental occurrence (such as the passing of a certain date), or even, as we argue, by an action of the liable party. We contend

[14] The implementation of the `all` operator is described in §A1.7 in the Appendix.

[15] Often, a deontic state-of-affairs such as being-obliged in terms of a particular clause may be brought about. However, the state of affairs need not necessarily involve obligations, prohibitions, or permissions. It may be, for instance, buying or owning in terms of a particular clause. The state of affairs brought about, though not necessarily deontic, is essentially a *legal* state of affairs: i.e. one recognised as such by some clause of a normative system.

that power and liability are in fact best treated using the single function device to describe how occurrences fitting a convention bring about other occurrences. Consider that an occurrence of violating a prohibition against interference may, through a function being_obliged_function2(...), bring about an occurrence of being-obliged (a so-called 'secondary' obligation) to pay damages. Neither Jones and Sergot [JS96] nor Daskalopulu [Das99] provide examples of cases where violations bring about obligations, since it would be strange in their terminology for a party to exercise their power by bringing about a violation, and in so doing bring about a duty upon themselves. Indeed, even for Hohfeld, this situation is awkward since *x* is both the liable (to have his legal relations changed) and the empowered (to change legal relations) party. Hohfeld says a person holding a 'power' has the legal ability by doing certain acts to alter legal relations; the one whose legal relations will be altered if the power is exercised is under a 'liability'. And yet here, *x* clearly has a power to alter legal relations (by interfering and violating the prohibition), and a liability to have his legal relations altered (by interfering). This situation is awkward because Hohfeld sees legal relations as being "relations of one individual with another", yet here *x* has a relationship with himself. As is evident from our function device, we do not see it as problematic that any occurrence, whether intentional or not, positive or negative, can bring about legal relations upon any person. The liable party is often a passive participant in an occurrence in the domain of the function which encodes the liability, but he may just as well be an active participant in the occurrence (as in the case of interfering) or not participate in the occurrence at all (for example, where an obligation is contingent on the occurrence of an environmental event).

The scope or longevity of rights (e.g. *one-shot* versus *persistent* rights) may be accounted for by limiting the domain of the function that brings about the legal consequences, through suitable query definitions:

5.5.1 One-shot (single-use) rights

☑
Requirement 34 (pg 70):
Both once-off and persistent rights should be expressible.

Ordinal queries (§A1.5) are well suited to capturing so-called 'one-shot' rights, such as the right to buy a certain quantity of a commodity, which may be exercised only once. In this case, it is only the *first* (in ascending temporal order) occurrence of purchasing that commodity that counts as a valid purchase. Formally, the domain of the *buying_function* here is an ordinal query returning only the first allocated (prima facie) purchase occurrence. 'One-shot' rights are contrasted to 'persistent' rights [NSJ98]; this is similar to the distinction between rights with 'cardinality' of actions and 'standing' rights (see page 70).

5.5.2 Persistent (multi-use) rights

In a competition that allows multiple entries per person, the right to enter, which may be exercised many times, is a persistent right. We could capture the right to unlimited entries using a (non-ordinal) query which returns *all* occurrences in the set, as opposed to an ordinal query which returns only the first *n* occurrences. All occurrences of entering are covered by the (non-ordinal) query, and would count as valid entries[16].

Cole, Derrick, Milosevic, and Raymond [CDMR2001] leave a treatment of action cardinality for future work. Norman, Sierra, and Jennings [NSJ98] capture their distinction between one-shot and persistent rights in a formal theory using dynamic logic. We implement one-shot rights as ordinal queries. The advantages of this approach over a dynamic logic mechanism is that ordinal queries generalize straightforwardly to two-shot, or *n*-shot rights, and furthermore allow for non-temporal ordering criteria when choosing which occurrences are *covered* (§3.3) by the right. The importance of non-temporal orderings is evident in health insurance scenarios, where a patient may have the right to two visits to their doctor in a given

[16] The right to vote, which some might think of as a persistent right, is actually many once-off rights. Each official election instantiates a once-off right: the voter is empowered to make only one valid vote per election. Successive votes do not count, and indeed, under some systems of law, successive attempts to vote will result in nullification of the first vote.

year. That is, only two visits are regarded as 'visits in terms of clause *x*' by the scheme. For patients that have undergone four treatments in a given year, generous commercial health providers might select the most costly visits as qualifying, irrespective of whether these were temporally the first two visits in the year or not. Dynamic logic cannot capture non-temporal modes of ordering, whereas ordinal queries incorporate these easily through specification of alternative comparator functions for use in ordering.

5.6 Obligations (Duties)

The notion of an obligation, or duty, is complex and multi-faceted. The aspects requiring treatment include obligation definition and fulfilment, violation and secondary obligations, obligation directedness, prima facie and all-things-considered obligations, conditional obligations, ought-to-do and ought-to-be obligations, several obligations, individual and collective obligations, and termination conditions.

5.6.1 Obligation definition and fulfilment

Take the obligation 'SkyHi is obliged *to pay Steelmans $25,000 by 1 September 2001*' (Clause C.1 of our application scenario). This embeds the query: 'select the first payment, before 1 September 2001, of $25,000 by SkyHi to Steelmans', since that is the nature of the obliged occurrence, and only occurrences of that nature can fulfil the obligation. The query asks for the *first* payment because it is exactly one payment that is obliged. The obliged occurrence may not exist yet, but when it does, we can determine that it was obliged if it fits the query. The obligation of SkyHi to pay Steelmans (Clause C.1) is given as being_obliged1 in Table 11, where Query19, whose representation was shown on page 84, is a pointer to a stored query that describes the set of obliged occurrences in the obligation, being_obliged1. The obligation of Steelmans to deliver 10 tons of steel to SkyHi (Clause C.2) is being_obliged2 in Table 11. It should be noticed that, like the obligation to pay, the obligation to deliver here is

an unconditional one, and that Steelmans is obliged to deliver even if SkyHi has not yet paid[17].

What it means to 'fulfil' an obligation was defined in Clause L.1 of the laws defined in our application scenario. If in the future the query associated with an obligation is 'filled' – that is, the count of items fitting the query is the maximum number of results the query can produce (one result, in our examples) – the obligation is said to be 'fulfilled'. Otherwise, the obligation is not (yet) fulfilled. Representation of obligation fulfilment is also shown in Table 11.

Occurrence	Role	Participant
being_obliged1	obliged	Query19
	IsAccordingTo	Clause C.1
being_obliged2	obliged	Query510
	IsAccordingTo	Clause C.2
fulfilling_function1	domain	Query511
	fulfilled	Query512
	IsAccordingTo	Clause L.1

Query19 (see page 84) = first (occurrence of SkyHi paying Steelmans $25,000 for the delivery, where paying is before 1 September 2001)

Query510 = first ((occurrence of delivering where (delivered is steel and participates in role item_measured in an occurrence of measuring where quantity_measured is 10 and unit_of_measure is tons)), intersection (occurrences of delivering which are before 1 October 2001))

Query511 = occurrences of count(results of |Query513|) being equal to max-possible-results(|Query513|)

Query512 = occurrences of being_obliged where (query counted in |Query511|) is obliged

Query513 (not shown in table) = queries in role obliged in occurrences of being_obliged

Table 11: Initial schema for storing obligations and their fulfilment conditions

As we shall shortly see, the initial schema in Table 11 for storing obligation instances requires a little fine-tuning to account for *allocation* of performances to obligations. To understand the problem, we now compare the mechanism of using identified occurrences for representing and assessing obligations, to the conventional

[17] Of course, the obligation to deliver may be voided if it is clear that SkyHi has no intention of ever paying. Voidance is dealt with in §5.6.9.

approach of using propositions and logical operators, and then refine our definition appropriately.

Standard Deontic Logic or SDL (§2.6) presumes that being obliged to do that which is prohibited is a logical contradiction. Obligation and prohibition are treated as operators, typically with the interdefinability axiom:

$$P\alpha =_{def} \neg O \neg \alpha$$

meaning that if a state-of-affairs, α, is permitted then it is not obliged that not α. In contrast, we take as our starting point the assumption that permissions and obligations are independent entities – variables that are quantified over – and that conflicting norms can exist. Our view is that being obliged to do that which is forbidden is not a logical contradiction, but a choice dilemma. The choice may involve deciding which identified directive to violate, or deciding which to regard as void in the circumstance (that is, in each case of application by a norm interpreter).

We need to distinguish the notion of an 'occurrence' from that of a 'proposition'. A proposition may be associated with a truth value: letting *A* be the proposition 'Steelmans delivered 10 tons of steel', we assess that *A* is true in the case that there are one or more occurrences of Steelmans having delivered 10 tons of steel. In contrast, an occurrence is an identified entity, which occupies a moment or interval of time and takes as its arguments participants in various roles. We might therefore have delivering1 which takes as its arguments consignment1 (which is measured as 10 tons of steel) in the role delivered and Steelmans in the role deliverer. Assuming SkyHi ordered 10 tons of steel on two separate occasions, following are two separate occurrences of Steelmans delivering steel:

delivering1 (on 15 September 2001) = (occurrence instance)

(role)	(participant)
delivered:	consignment1 (measured as 10 tons of steel)
deliverer:	Steelmans

delivering2 (on 20 September 2001) =

delivered:	consignment2 (measured as 10 tons of steel)
deliverer:	Steelmans

The proposition *A* ('Steelmans delivered 10 tons of Steel') is strictly true from 15th September onwards, since from that moment on it is true that Steelmans has at

some stage delivered the requisite amount of steel. A proposition, then, becomes true as a result of one or more identified occurrences. Of identified occurrences we can say only that they happened (or did not): we speak of their existence, rather than of their 'truth'. Since occurrences occupy (perhaps unspecified) moments or intervals of time, they are inherently temporal in nature, whereas propositions – in the absence of extension to cater for time – are atemporal.

We believe the notion of an occurrence, absent from standard treatments of deontic logic, is useful for the representation and assessment of obligations. Brown [Bro2000] speaks of the distinction between simply dischargeable obligations and standing obligations. The latter have been the traditional purvey of deontic logic. As we will illustrate here, it seems that Standard Deontic Logic copes poorly with simply dischargeable obligations. This is problematic since such obligations form a large portion of the obligations we wish to reason about in commercial contractual scenarios. Consider the case where SkyHi has ordered 10 tons of steel for the second time on 1st July 2001 and Steelmans has an obligation – a specific obligation – to deliver it. Taking *A* as the proposition that Steelmans delivered 10 tons of steel, in Standard Deontic Logic we might then say:

OA

meaning 'Steelmans is obliged to deliver 10 tons of steel'. How though, do we distinguish this second obligation, from Steelmans' previous obligation, which arose when SkyHi first ordered steel? In SDL, the first obligation is also represented as:

OA

Collapsing the obligations together through logical derivation results in dangerous information loss, since originally we had two obligations, but standard propositional logic leaves us with one:

$OA \wedge OA$

————— *(standard propositional logic)*

OA[18]

[18] Similar information loss occurs when we derive such conclusions as OA from $O(A \wedge B)$ in SDL. As SDL does not identify the particular obligation $O(A \wedge B)$ we have no way of determining that OA is in

SDL fails to distinguish between propositions about norms, and the actual norms themselves. Makinson [Mak99] comments that simply describing norms as true or false is insufficient, and that it is a fundamental problem of deontic logic that norms are simply considered to have truth values. We argue that, since propositions about norms are derived from the norms themselves, invalid or misleading inferences will result if we deal merely with the propositions rather than with the identified norms that make those propositions true or false. Assume, on both occasions, Steelmans fulfils their obligation to deliver 10 tons of steel (that is, A is true), we have:

$$\frac{OA \wedge A \wedge OA \wedge A}{OA \wedge A}\ (\textit{standard propositional logic})$$

The two different fulfilment occurrences are not distinguished here. Most problematically, it seems that, had Steelmans delivered the steel only the first time we would still have:

$$OA \wedge A$$

This says that A was obliged and A was done. Using Anderson's reduction [And58] of obligation to violation in the case of falsity of the obliged proposition ($OA =_{def} \Box(\neg A \rightarrow \textit{Violation})$), Steelmans' violation of their second obligation is not evident since A is true by virtue of fulfilment of the first obligation.

To rectify the above problems, it seems that norms (and occurrences which fulfil or violate those norms) should be individually identified. A treatment of obligations as entities, rather than as operators on propositions, has been proposed in Kimbrough [Kim2001]. Kimbrough has recommended identifying both obligation states (instead of the O operator) and violation states (instead of the insufficiently specific *Violation* predicate) to correct some existing deficiencies of SDL. We wish to justify Kimbrough's suggestion and, using our own notation, illustrate their pertinence to the multiple-steel-orders example.

---

fact a partial description of $O(A \wedge B)$. Maintaining an identifier for the obligation (e.g. using subscripts, $O_1(A \wedge B)$ and $O_1(A)$) would be useful in a database environment to allow us to look up the full description of the obligation O_1 when we have partial information on it.

In our world of identified norms, the particular obligation of Steelmans to deliver steel by 1st October may be denoted as being-obliged2, as shown in Table 11 (page 143). The second obligation may be represented similarly as, say, being-obliged20.

Now, using the simplified representation of delivery obligations from Table 11, our continuous query mechanism (§3.3) determines that delivering1 fulfils both the first obligation (being-obliged2) and the second (being-obliged20). This is because delivering1 ***fills*** the queries associated with both obligations (as both queries only ever return a maximum of one occurrence – hence the criterion 'first' in each query) and thereby ***fulfils*** both obligations.

Clearly, an adjustment is required to cater for the fact that the same performance is not intended to fulfil multiple obligations. Kimbrough [Kim2001] and Daskalopulu and Sergot [DS2002] recommend the use of a *sake_of()* predicate to allocate fulfilment occurrences to the obligations they are intended to fulfil. Since different allocations are possible over time, and depending on which allocation basis is used (e.g. most-recent-first, least-recent-first, or arbitrarily complex allocation criteria), we choose to use an occurrence of allocating in place of Kimbrough's *sake_of()* predicate. We might then represent the two obligations as shown in Table 12 below; the required amendments to the queries are shown in **bold**.

Occurrence	Role	Participant
being-obliged2	obliged	Query510
	IsAccordingTo	Clause C.2
being-obliged20	obliged	Query514
	IsAccordingTo	Clause C.2

Query510 = first ((occurrence of delivering where (delivered is steel and participates in role item_measured in an occurrence of measuring where quantity_measured is 10 and unit_measured is tons)), intersection (occurrences of delivering which are before 1 October 2001) **intersection (participants in role allocated in occurrences of allocating where allocatedTo is being_obliged2 and allocationBasis is FIFO)**)

Query514 = first ((occurrence of delivering where (delivered is steel and participates in role item_measured in an occurrence of measuring where quantity_measured is 10 and unit_measured is tons)), intersection (occurrences of delivering which are before 1 October 2001) **intersection (participants in role allocated in occurrences of allocating where allocatedTo is being_obliged20 and allocationBasis is FIFO)**)

Table 12: Corrected schema for storing obligations: with performance allocation constraints added

We then require a rule that each delivery of steel must be allocated, using some specified basis, to a single obligation. Following application of such a rule, we might have the following occurrences of allocating:

allocating1
- allocated: delivery1
- allocatedTo: being_obliged2
- allocationBasis: FIFO (first in, first out)

allocating2
- allocated: delivery2
- allocatedTo: being_obliged20
- allocationBasis: FIFO (first in, first out)

We can then ensure that each delivery satisfies only a single obligation. We do not here deal with assignment of performances to multiple obligations, such as when a single delivery of 20 tons of steel (or similarly, a single payment of many dollars) fulfils multiple obligations. In the latter case, each delivered *ton of steel* (or similarly, paid dollar) is allocated to an obligation, rather than each *delivery* of tons of steel (similarly, payment of dollars). Neither do we deal with the complexities of accumulation of debts, such as when multiple purchases-on-account during a month are aggregated at month end into a single obligation to pay during the following

month, and such obligations may accumulate from month to month. Assignment of payments to purchases, and corresponding conclusions about transfer of ownership for each item purchased during the period are generally controlled by sophisticated organization-specific policies, which are outside the scope of this work. We make the simplifying assumption that each discrete performance occurrence pertains to a single obligation.

Given our above-specified obligations, and our ability to deduce their fulfilment by determining whether queries are filled, we can derive the following occurrences when Steelmans delivers 10 tons of steel on time, on both occasions:

fulfilling1
    fulfilled: being_obliged2
    fulfiller: delivering1
fulfilling2
    fulfilled: being_obliged20
    fulfiller: delivering2

To illustrate our definition of obligations graphically, we might think of a metaphor of eggs and buckets. Each time an observable occurrence happens, create an egg to describe that occurrence (e.g. delivering1, paying1). Each time an obligation is incurred, create a bucket to represent what is obliged under that obligation[19]. Only eggs of a certain description can go into the bucket. Assess each obligation by checking, before or after the deadline, whether the bucket is full. Using the egg and bucket metaphor, Figure 9 shows an obligation to deliver that has not yet been fulfilled or violated, whilst Figure 10 shows an obligation that has been fulfilled.

[19] Note that when an obligation is incurred you create both an egg and a bucket since the obligation is itself an observable occurrence to be recorded, and therefore justifies an egg.

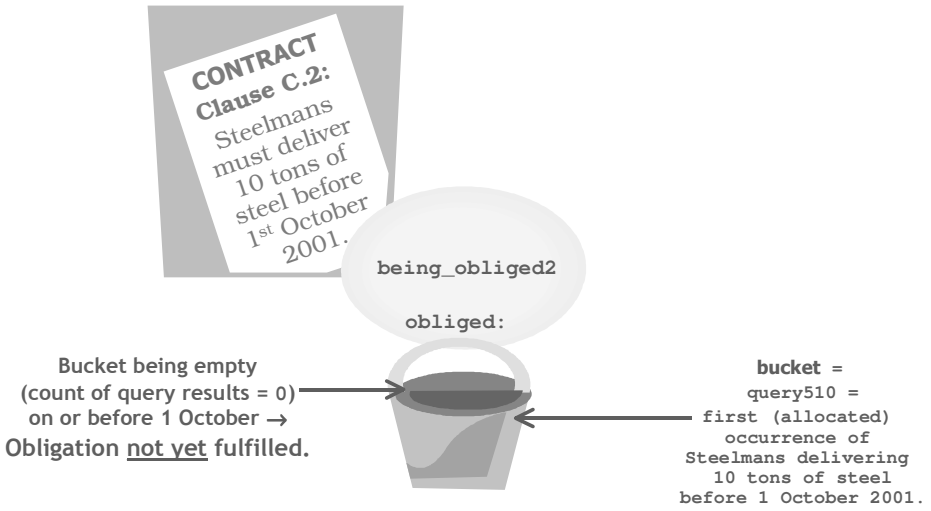


Figure 9: A not-yet-fulfilled-or-violated obligation

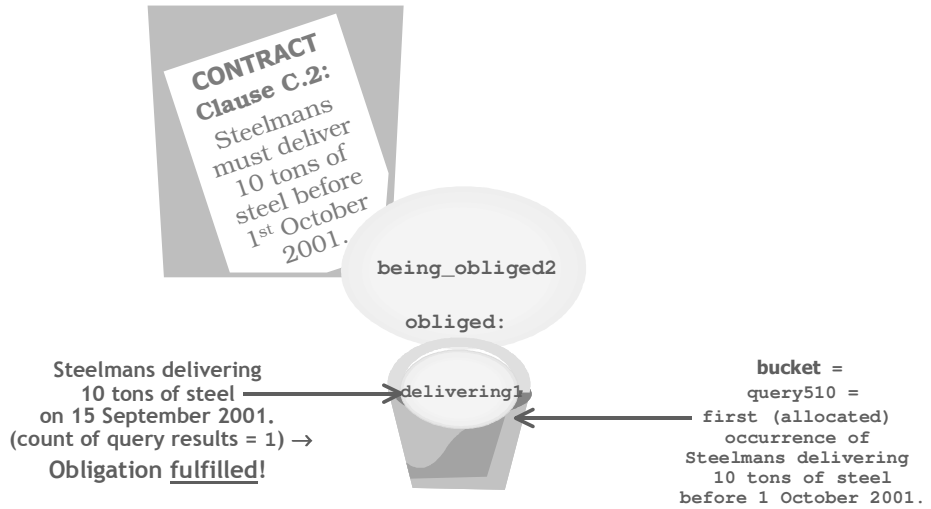


Figure 10: A fulfilled obligation

The query-based approach is well-suited to the specification of free-choice obligations such as the obligation to 'deliver a thick-base pizza or a burrito with extra cheese'. Such an obligation is awkward for an approach based on matching prototypical objects (e.g. [DS2002]): it is unclear what the prototypical object in this case is, since pizzas and burritos are different object types entirely. A query-based

approach simply states the query as 'first of ((delivering a thick-base pizza) union (delivering a burrito with extra cheese))'. As any appropriate delivery can fill this query, the obligation is straightforward to express.

It is evident that the identification of obligations and occurrences brings some benefits over the operator-based account of Standard Deontic Logic: separate obligations can be uniquely identified, and we can determine which of these several obligations has been fulfilled, and by which occurrences of delivering, even when the content of the obligation (e.g. 'delivering 10 tons of steel') is similar.

5.6.2 Violations: Primary and secondary obligations

The obligations, being_obliged1 and being_obliged2, of SkyHi and Steelmans respectively, are *primary* obligations; their appearance is *not* contingent on the existence of violations. In contrast, *secondary* obligations are brought about by violations. Secondary obligations typically impose duties upon the violator. Clause L.3 of our application scenario tells us there are secondary obligations to pay damages for violation. Often the violator is not *personally* liable for the violation – for example, in the case of insurance or other transfer of liability – as the secondary obligations may fall upon other parties. So, the secondary obligations may describe occurrences where the agent (i.e. duty-bound party; e.g. payer or deliverer) is the original violator, or where the agent is some other party.

In order to represent secondary obligations, occurrences of violating must be stored. According to the definition of 'violating' given in Clause L.2 of our application scenario, if too few items (less than one in our examples) fit the query after the deadline (1 September 2001 in Clause C.1, 1 October 2001 in Clause C.2), the obligation is violated[20]. Table 13 gives a representation of how violations are

[20] Note that if the count, before the deadline, of items fitting the query is zero, the obligation is not violated, but rather 'not (yet) fulfilled' (which is not to imply that it ever will be fulfilled). The Disquotation Theory models obligation violation, but not fulfilment, explicitly, and therefore the subtlety of 'not violated' meaning either 'not yet fulfilled' or 'fulfilled' is not captured.

It should be pointed out, also, that we see occurrences as being of a single type, implying obligation states and violation states are distinct entities. Kimbrough argues that an obligation state, ought(e_1), can be the same as a violation state, violating(e_1). Our contention is that each is an

produced. The table also illustrates that violations bring about liability to damages (obligations to pay) through a function that creates secondary obligations (Clause L.3). Figure 11 shows the assessment of obligation violation graphically, using the egg and bucket metaphor introduced on page 149.

Violations of individual terms may bring about so-called 'breach of the contract'. It should be noted that not all violations or breaches are punishable. Even if secondary obligations are entailed, these secondary obligations may be prima facie obligations (§5.6.4) that are voided (§5.6.9), thereby forgiving the transgression.

Occurrence	Role	Participant
violating_function1	domain	Query515
	violated	Query516
	IsAccordingTo	Clause L.2
being_obliged_function2	domain	Query517
	obliged	Query518
	IsAccordingTo	Clause L.3

Query513 = see page 143

Query515 = occurrences of count(results of |Query513|), after their deadlines, being < max-possible-results(|Query513|)

Query516 = occurrences of being_obliged where (query counted in |Query515|) is obliged

Query517 = occurrences of successfully instigating prescribed procedure following occurrences of violation of occurrences of being-obliged

Query518 = first occurrence of paying damages for the violation for which legal action was instigated in |Query517|

Table 13: A schema for storing violation conditions, and consequent liability (secondary obligations) to pay damages

independent entity: an occurrence, being_obliged1, and an occurrence, violating1, where the participant in the role violated in violating1 is the obligation being_obliged1.

Finally, it is worthwhile to point out here that the law may more generally define violation as coming about whenever there is an occurrence of the obliged state-of-affairs *being_impossible*., and the duty-bound party is to blame for this. Violation would then be defined as a function whose range is relevant occurrences of *being_impossible*. Occurrences of *being_impossible* are brought about by occurrences of counting, where the obliged occurrences are counted, the count is zero, and the count was made after the deadline. However, occurrences of *being_impossible* may also be brought about by other wilful acts or negligence before the deadline. The definition of violation in terms of occurrences of *being_impossible*, captures the notion in law that it is possible to violate an obligation *even before the deadline*, by doing – or, indeed, omitting – something, and making it impossible to fulfil the obligation.

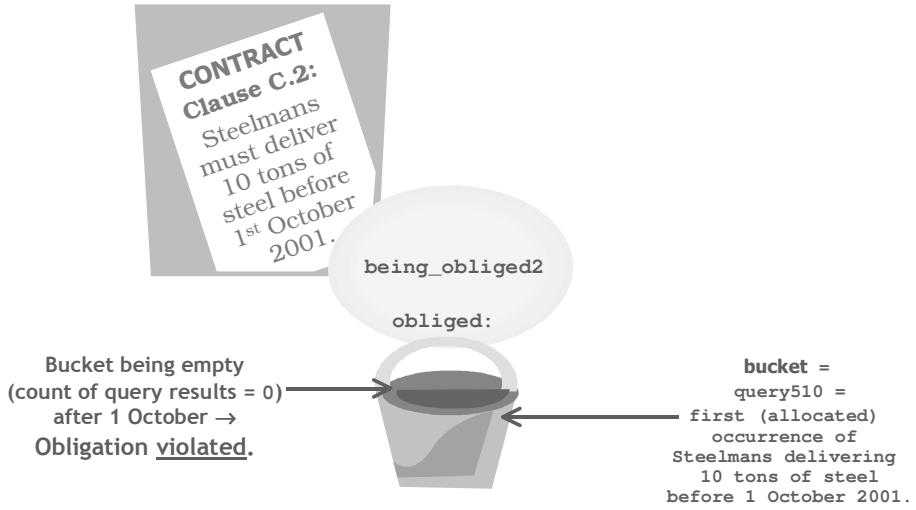


Figure 11: A violated obligation

What prohibition is the obligation to pay or deliver by a deadline equivalent to? It might be tempting to think of the obligation to pay by 1st September 2001 as equivalent to the prohibition against paying after 1st September 2001. However, we must be careful of our reading and implementation of this prohibition. We do not want a prohibition against *paying* after 1st September 2001, since this would fire a violation each time a payment is made after 1st September 2001, which is not what we intend. 'Better a late payment than no payment' is not captured by that logic interpretation. Rather, our prohibition is against *the single case where* no suitable payments are made before 1st September, and the violation conditions for this have been defined already in violating_function1.

5.6.3 Directed obligations

With particular reference to the obligation being_obliged1 in Table 11 and Table 13, we have specified the following pertinent parties:

- party responsible to act (duty-bound party): payer mentioned in Query19 describing the primary obligation (SkyHi)
- party responsible in surety: payer mentioned in Query518 describing the secondary obligation (happens to be SkyHi again)
- party directly benefiting from the obliged action: payee mentioned in Query19 describing the occurrences obliged under the primary obligation (Steelmans)
- party empowered to initiate recourse: instigator mentioned in Query517 describing the occurrence that brings about the secondary obligation (happens to be Steelmans again)
- party entitled to outcome of recourse: payee mentioned in Query518 describing the secondary obligation (Steelmans, again)
- party issuing the norm: the party uttering, or organization or document associated with, the clause (for example, the Clause C.1 is associated with the contract between SkyHi and Steelmans)

Disquotation Theory says nothing specific about responsibility in surety, empowerment to initiate recourse (suggested by Makinson [Mak86]), and entitlement to the outcome of resource. Simply adding additional thematic or domain-specific roles to the primary obligation to capture these role-players is inappropriate. Instead, we specify what additional obligations and powers come about upon violation of the primary obligation.

☑

Requirement 31 (pg 67): Directed obligations, with actor, beneficiary, liable party, source utterance, and issuer, must be expressible.

5.6.4 Prima facie (defeasible) and all-things-considered obligations

As each obligation is *according to a particular clause*, these obligations may be considered as a representation of the notion of *prima facie* obligations discussed in the deontic logic literature [PS97]. To avoid confusion, we ought to note at this point the distinction between *primary* obligations and *prima facie* obligations put forth in Prakken and Sergot [PS97]. There is one contrast between primary and secondary obligations (the latter coming into being upon violation of the former), and another contrast between prima facie and all-things-considered obligations (the former being on-the-surface, if clauses are considered in isolation, and the latter being what is said by the contract, act, or law as a whole). Our treatment of obligation instances as being 'according to a particular clause' and our selection of particular clauses to void during conflict resolution (described in detail in Chapter 6) is intended to deal with the distinction between prima facie (defeasible) and all-things-considered obligations (see also §5.6.9). The distinction between primary and secondary obligations, we have seen (Table 13), is dealt with through occurrences of violating and functions that bring about secondary obligations as a result of those occurrences of violating.

5.6.5 Conditional obligations

Obligations, like other occurrences, are often contingent: they (i.e. occurrences of *being-obliged*) may be triggered by some uncertain future occurrences. We have seen (§5.6.2) that conditional obligations, such as secondary obligations, may be represented through the function device, where the contingent occurrences are the domain of the function that produces individual obligation instances.

Consider the contingent obligation, specified in Clause C.4 of our application scenario, to refund the amount paid upon a return valid in terms of Clause C.3. Table 14 depicts the representation of this contingent obligation as being_obliged_function3, which produces an occurrence of being_obliged for each occurrences of 'returning, according to Clause C.3'.

Occurrence	Role	Participant
being_obliged_function3	domain	Query519
	obliged	Query520
	isAccordingTo	Clause C.4

Query519 = occurrences of returning where participant in role isAccordingTo is Clause C.3

Query520 = first occurrence of refunding where (refunded is SkyHi) and (participant in role refunded_amount is (total of participants in role paid_amount in (occurrences of paying where (participant in role paid_for is (participant in role returned in |Query519|)))))

Table 14: A schema for storing a conditional obligation

☑

Requirement 24 (pg 44):
We must be able to refer to individual obligations of each party, and trace each obligation to the general prescriptions, and events, that brought it about, or that terminated it.

From the schemas in Table 10 (encoding a right to return) and Table 14 (encoding a claim contingent on valid returns), it should be evident that an occurrence, returning1, which is a physical return within 30 days of purchase, brings about a notional return, say returning11, which is a valid return in terms of Clause C.3. returning11, in turn, brings about a claim, say being_obliged31, which is an obligation to refund the amount paid. The return, returning11, synonymous with returning_function1(returning1), and the obligation, being_obliged31, synonymous with being_obliged_function3(returning11), can be stored as follows:

returning11 = (legal consequence)

source_rule:	returning_function1	(provenance)
source_occurrence:	returning1	(evidence)
returned:	steel purchased in purchasing1	
returner:	SkyHi	

being_obliged31 = (legal consequence)

source_rule:	being_obliged_function3	(provenance)
source_occurrence:	returning11	(evidence)
obliged:	Query521	

Query521 = first occurrence of refunding total amount paid for purchasing1

As illustrated in Figure 12 and Figure 13 below, the birth of obligation instances from general obligation policies and specific occurrences, may be seen as analogous to the birth of a child from parents. In both cases, a genetic marker – DNA for children, source identifiers for obligations – allows us to trace the product to its origin.

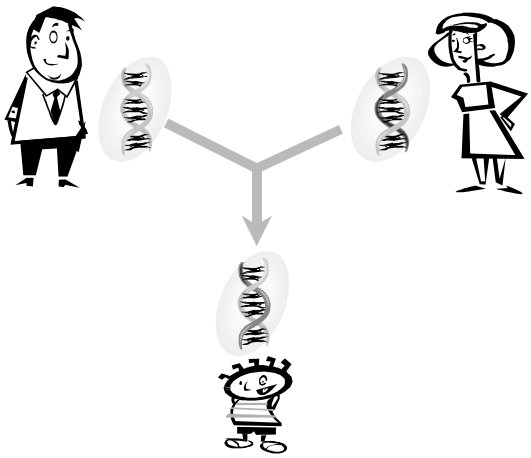


Figure 12: Birth of a child from parents

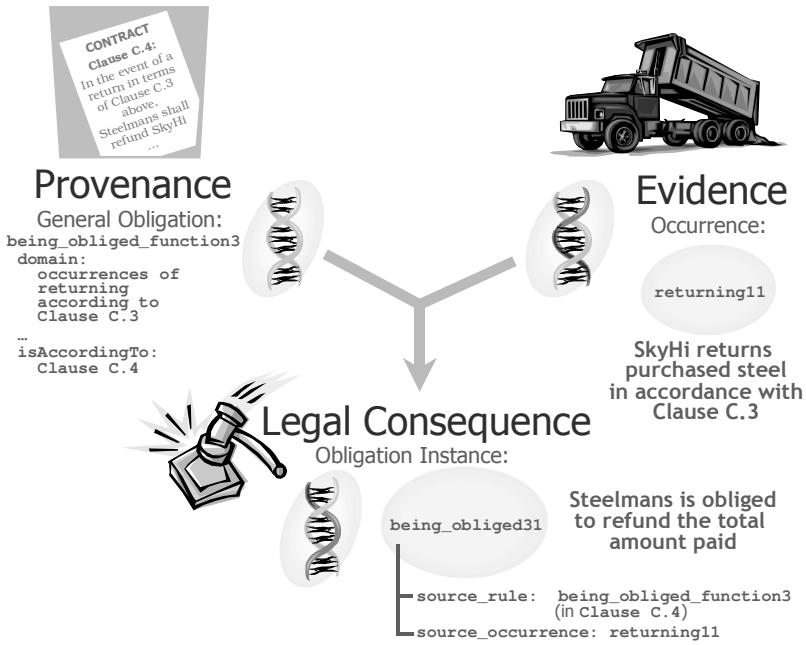


Figure 13: Birth of an obligation instance from policy and evidence

5.6.6 Ought-to-do and ought-to-be obligations

☑ **Requirement 22 (pg 43):** It must be possible to express and monitor both obliged actions and obliged states of affairs.

An approach based on the occurrence abstraction provides a uniform treatment of events and states for obligation fulfilment monitoring. The status of both ought-to-do and ought-to-be obligations can be assessed by counting how many occurrences of the desired event or state (e.g. action, process, or state-of-affairs) have occurred by the deadline, and by triggering violations when a desired event or state does not obtain. Earlier policy-based approaches associate obligations solely with method invocations (see page 42). Such approaches cater for push-based calls-to-action. However, they neglect the need for execution components to be able to assess whether desired states-of-affairs have obtained in a pull-based manner and, by hard-wiring method calls, can restrict components from pursuing alternative means of obligation fulfilment that were not originally available.

5.6.7 Several (multiple) obligations

☑ **Requirement 32 (pg 67):** Obligations should be individually identified.

A particular English sentence (clause) may define a single obligation, such as the single occurrence of *being-obliged (by that clause)* to pay Steelmans $25,000 by 1 September 2001. However, a single clause may also define how multiple separate obligations are brought about. Consider Clause L.3 of our application example, which effectively says that 'each occurrence of successfully instigating the prescribed procedure after each violation of an obligation brings about a separate obligation to pay damages'. Interpreting Clause L.3 as implying the existence of a single occurrence of being obliged would be incorrect, as this would imply a *joint* or *collective* obligation by all liable parties together to pay damages to all aggrieved parties together. Rather what is intended is that there are several, *separate* obligations. The several obligations may be formally defined as:

f : occurrences of successfully instigating → occurrences of being-obliged

This reads that a function *f* maps occurrences of successfully instigating to occurrences of being obliged. In Table 13, *f* is the identified function being_obliged_function2. being_obliged_function2 takes as its domain occurrences of successfully instigating and produces occurrences of being-obliged. For example **instigating1**, a successful instigation, generates the particular obligation being_obliged_function2(**instigating1**) where being_obliged_function2 was a function defined in Clause L.3. It should be clear from this discussion that, in the case of clauses that generate multiple obligations, individual obligations can be identified by combining the function identifier of the general obligation (e.g. being_obliged_function2) with the identifier of the occurrence that generated the obligation. being_obliged_function2(instigating1) is synonymous with the following specific obligation instance that may be explicitly stored in the occurrence store:

being_obliged21 =		(legal consequence)
source_rule:	being_obliged_function2	(provenance)
source_occurrence:	instigating1	(evidence)
obliged:	Query522	

Query522 = first occurrence of paying damages for the violation for which legal action was instigated in instigating1

being_obliged21 (where 21 is a unique identifier) is another name for the particular obligation instance brought about from the general obligation policy (being_obliged_function2) in the light of specific occurrences (instigating1). The link between an obligation policy (function), a happening, and the resultant obligation instance is recorded by storing the sources of the obligation as its source_rule and source_occurrence attributes.

☑

Requirement 27 (pg 50): Existence of obligations must be derived from interpretation of law, rather than from a closed set of communicative acts or a rigidly defined protocol.

Query522 was formed through the instantiation of bound variables defined in the general obligation. In Table 13 (page 152) we have defined the domain of the function using a query, Query517, and specified the participants in the occurrences in the range of the function. Bindings from a participant in an output occurrence to a participant in the corresponding input occurrence that produced that output are

represented using |Query517| embedded in any valid query expression. |Query517| in the query is then substituted with the corresponding value from the domain of the function. For example, assume instigating1 is a successful instigation. Then, Query518 (first occurrence of paying damages for the violation for which legal action was instigated in |Query517|), generates Query522 (first occurrence of paying damages for the violation for which legal action was instigated in **instigating1**), for the occurrence being_obliged_function2(**instigating1**), also known as being_obliged21, brought about by the occurrence **instigating1**.

The representation of several obligations is not dealt with in Lee's augmented Petri Nets [BLWW95] and Daskalopulu's state machine based [Das99] approaches to monitoring obligations, which we reviewed earlier (§2.5.7). In those approaches, the single obligation of a client, Peter, to pay a supplier, Susan, can be denoted. However, the approaches are not able to capture more embracing legislative provisions such as 'clients are obliged to pay suppliers for purchases' where a single sentence defines several obligations. In this case, each occurrence of purchasing brings about, via a function, an obligation (occurrence of being obliged) to pay for that purchase.

5.6.8 Impersonal and collective obligations

☑

Requirement 21 (pg 42):
Impersonal (collective) obligations must be expressible and checkable.

Functions may be used to represent the generation of individual, separable obligations on each member of a group. However, equally important, and frequently mentioned in the deontic logic literature, is the ability to represent *impersonal* and *collective* obligations, where no specific individual bears the personal responsibility. Moffett and Sloman [MS93] mention the notion of collective responsibility, but no implementation is provided in the subsequent work on Ponder [DDLS2001]. Ponder distributes an obligation to *all* subjects of the obligation – that is, distributes multiple, separate obligations – whereas a collective obligation is meant to capture the idea that the obligation is one obligation and *any* of the members of the group could fulfil the obligation. Only a

single occurrence, by any one of the group members responsible, needs to be brought about to fulfil the collective obligation.

Consider the obligation, say being_obliged4, of any report-writing component to generate year-end reports on 30 September 2001. Assume there are many such components able to fulfil the obligation. In our approach, the obliged occurrence can be described using the query, say Query523, which returns the first occurrence of generating, on 30 September 2001, year end reports, by a report-writing component. Certainly, there is some indeterminacy here, as any of a number of components could fulfil the obligation[21], but this is not problematic since we require only that the obligation be fulfilled by some member of a group and do not prescribe who specifically must fulfil it. Our high-level requirement need not be polluted by specific indication of which particular component is responsible, but can nevertheless meaningfully specify responsibility of a group. Our representation is able to assess when the group's responsibility has been fulfilled by determining when the query, Query523, is filled.

We of course assume that all components are diligent – i.e. that one of them will take up, or be assigned, the task. And in the event that none do, we are equally able to assess that the group's obligation has been violated, since no occurrences fitting the query described in the obligation have occurred by the deadline.

5.6.9 Life cycle: From birth to termination

We have seen how the function device may be used to explain how obligations *arise* or are *incurred* (§5.6.2, §5.6.5). Parties may also define arbitrary conditions that specify when obligations – or indeed other types of legal relations – *terminate*. These are called 'termination provisions' [TB99, p146]. For example, the obligation to repay a government-granted student loan typically ceases in the event of death, retirement, or (some types of) disablement. 'Exclusion'

☑

Requirement 33 (pg 70): Obligation life-cycle must be modelled. Obligations must be traceable to the events, states, and regulations that brought them about or cancelled them out.

[21] Though we do not deal formally with such matters here, some co-ordination mechanism is obviously desirable to ensure that no component tries to fulfil the obligation when another component is busy with it, and looks likely to succeed.

clauses [TB99, p147], amongst which are 'force majeure' provisions [TB99, p149], define the circumstances under which an obligation to pay damages becomes voided, thereby releasing (excluding) a party from liability. Legislation on 'frustration' [TB99, p152] also defines circumstances under which provisions become voided. An obligation is 'frustrated' when its performance is impossible, illegal, or pointless. Let us now look in more detail at the end of the life cycle of norm instances.

In Standard Deontic Logic (page 63) it is not clear that Steelmans' first obligation to deliver the steel was fulfilled: $OA \wedge A$ should, intuitively, imply $\neg OA$ since a fulfilled dischargeable obligation no longer stands. Or more specifically, we should be able to infer that the obligation, being_obliged2 (Table 11, page 143), once was in force, but is now fulfilled and no further call to action results. We therefore must be aware of occurrences of either its fulfilment, violation, or being voided. For example:

```
fulfilling1
    fulfilled:          being_obliged2
violating1
    violated:           being_obliged2
being_void1
    voided:             being_obliged2
```

Capturing such occurrences allows us to capture the *life-cycle* of an obligation, and thereby assess whether it is still active and requires fulfilment. Standard Deontic Logic allows us to make no such inferences. This is because SDL deals with truth-valued propositions about general standing obligations, rather than with specific identified obligations.

As we have seen, there are many ways in which an obligation may terminate: each of the above occurrences (fulfilling, violating, being-voided) counts as a cessation of the obligation[22]. We can define a function – shown in Table 15 below – to capture this.

☑

Requirement 23 (pg 43):
The implementation must allow introduction of broad reaching provisions (e.g. defining 'fulfilment' and 'violation') by a single insertion anywhere in the specification.

Occurrence	Role	Participant
ceasing_function1	domain	Query524
	ceased	Query525

Query524 = (occurrences of fulfilling) union (occurrences of violating) union (occurrences of being_void)

Query525 = participants in role theme[23] in |Query524|

Table 15: Defining termination or cessation of an obligation

A fulfilment, fulfilling1, of say being_obliged2, would then produce the following cessation occurrence:

ceasing1 =		(legal consequence)
source_rule:	ceasing_function1	(provenance)
source_occurrence:	fulfilling1	(evidence)
ceased:	being_obliged2	

[22] In contrast, CANDID (page 61) recognizes only fulfilling and reneging as ways in which an obligation may terminate [Lee80, p97]. As we shall see later (Chapter 6), accepting voidance as a means of obligation termination is essential for conflict resolution.

[23] theme here refers to any of the open set of role names: fulfilled, violated, voided since these domain-specific roles can be generalized to the semantic role 'theme' commonly used in knowledge representation in artificial intelligence [All95; Sow2000].

The notion of what it means to be a *current* obligation may be defined by the following laws pertaining to obligation life cycle:

Commercial Trade Act

…

A party is only obliged to fulfil obligations that have not ceased. **Clause L.4**

…

Using Clause L.4 as our basis, a 'current obligation' can then be defined as illustrated in Table 16 below.

Occurrence	Role	Participant
being_obliged_function4	domain	Query526
	obliged	Query527
	isAccordingTo	Clause L.4

Query526 = (occurrences of being_obliged) minus (participants in role ceased in occurrences of ceasing)

Query527 = participants in role obliged in |Query526|

Table 16: Defining 'current' (active) obligations

We might take obligation instances generated by being_obliged_function4 as *all-things-considered* obligations (§5.6.4), since the function takes as its domain *prima facie* obligations, but only outputs obligation instances (occurrences of being_obliged) that have not been voided by some other law. It should be noticed that the function device provides us with a means of *selecting appropriate interpretations* as to what is obliged; see also page 105 for discussion of some ways of selecting provisions.

Figure 14 and Figure 15 below depict how the life-cycle of an obligation instance is in some senses comparable to the life-cycle of an individual person.

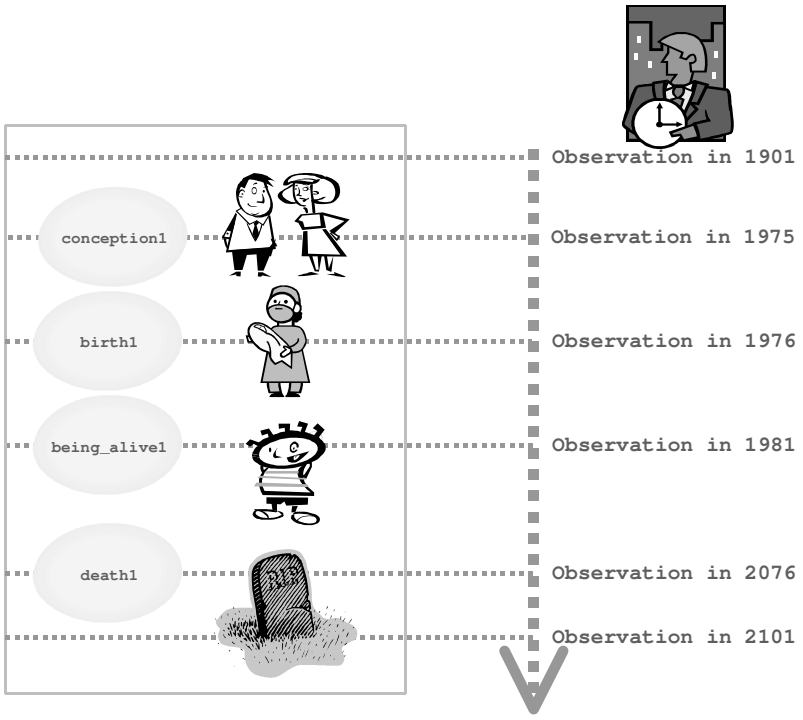


Figure 14: Life cycle of a person

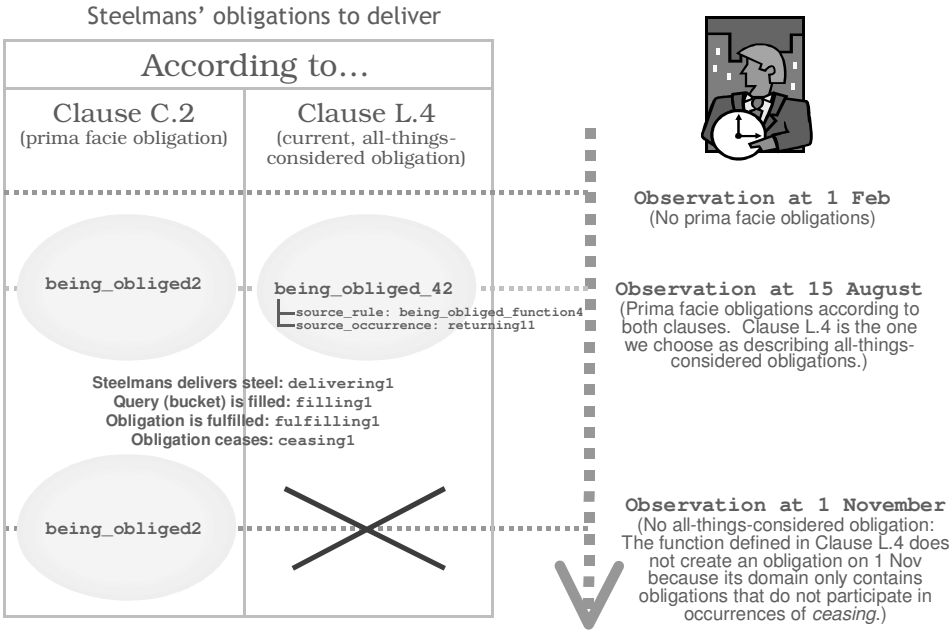


Figure 15: Life cycle of an obligation instance

5.7 Summary

We saw in this chapter how a practical extension of Kimbrough's Disquotation Theory provided us with a means by which provisions could be represented and stored. We showed how finding the domains an occurrence is in – what queries cover it – allows us to assess which provisions are *applicable*, and generate *legal consequences*, in a given situation. We also illustrated how we determine when a provision is *fulfilled* or *violated*. Contractual notions such as defined terminology, powers, authority, duties, and breaches were catered for through stored functions, and occurrences of prohibiting, permitting, obliging, fulfilling, and violating. Conditional obligations were supported through functions. In addition, the function device allowed the expression of provisions where multiple (several) obligations are brought about by separate occurrences – a problem not tackled in augmented Petri Net- and state machine-based mechanisms (§2.5.7), which deal with provisions referring to single obligations. Subjective interpretations, as would eventuate from alternative institutional perspectives and systems of law, were supported through function and clause identifiers.

☑ **Requirement 30 (pg 62):** An approach is needed where provisions are explicitly captured as data, and are thus readily available for inspection and analysis.

☑ **Requirement 26 (pg 49):** Provisions (legal relations), should be explicitly stored, not implicitly encoded in process models.

Importantly, by integrating jurisprudential theories from Hohfeld, Anderson, Bentham, Kimbrough, Makinson, Jones, and Sergot, we represented in this chapter various subtleties of the phrase 'having a right to …' in English. We saw that 'your having a right to …' may mean that:

- a vested liberty exists (5.4.1), where others are prohibited from interfering with a state of affairs, such as the state intended to be achieved by an obligation.
- your performing a specified action does not bring about any violations (5.4.2).

☑ **Requirement 11 (pg 32):** Fundamental legal conceptions – such as duties, privileges, powers, and immunities [Hoh78] – must be natively incorporated in the development approach.

- you possess the power to bring about a state of affairs (§5.5).
- you are the party expected to benefit from the performance of an obligation (§5.6).
- you are the party entitled to the outcome of recourse if the obligation is not fulfilled (§5.6).

Clearly, a complex web of interleaving provisions is easily woven. Furthermore, in dynamic environments where provisions are frequently added by parties acting semi-autonomously, it often happens that a situation is covered by multiple, conflicting provisions. Mechanisms for conflict detection and resolution are discussed in the next chapter.

☑

Requirement 28 (pg 51):
The model should provide fundamental legal conceptions, rather than hard-code the constraints of a particular system of law.

Chapter 6

Conflict Expression, Detection and Resolution

In environments where norms are specified by multiple semi-autonomous norm-givers, conflicting provisions are inevitable. Advanced facilities for detection and resolution of conflicts between legal provisions are therefore critical for an application development approach that relies on the storage of inter-related clauses from company policies, contracts, and regulations.

We suggest that, for conflict *expression* (§6.1), individual norms can be seen as *situated*: they are generated from defined functions and must be tagged with the context in which they were written or spoken. Conflict between norms can be *detected* by determining overlaps between obliged, permitted, and prohibited occurrences (§6.2). Conflict *resolution* (§6.3) can be achieved by voiding norm instances of particular origin and selecting those current obligation, permission, and prohibition instances that remain. Alternatively, we may simply decide to violate some of the conflicting norm instances in order to fulfil others. We tag our conclusions with a *time* (§6.4), so that, without contradiction, we may non-monotonically conclude different results about which obligations, prohibitions, or permissions stand undefeated as cases and norms vary over time.

6.1 Expressing Conflict: Identity & Situation

We have argued (Chapter 5) for a treatment of obligations, permissions, and prohibitions that differs from the standard treatment of these notions in deontic logic. We have proposed that instantiated norms be treated as individual, **identified** entities – that is, variables that can be quantified over – rather than simply as logical operators as in Standard Deontic Logic. This allows us to refer to *specific instances* of obligations (Table 13, page 152; Table 14, page 156; Table 16, page 164), permissions (Table 9, page 136), and prohibitions (Table 6, page 131). We suggested that norms take, as their arguments, descriptions of sets of occurrences, rather than simply propositions as in the standard treatment. We provided a detailed account of the *life-cycle* of norms (§5.6.9): having explained how individual identified norm-instances are generated from general norms through functions of occurrences, we looked at how each such instance's life may end with its fulfilment, violation, or nullification.

☑

Requirement 12 (pg 32):

Rule attributes – such as author, specification time and document or utterance location, scope, and jurisdiction – should be stored, as should attributes of entities related to rules – such as author's roles over time.

Derivation of specific identified obligations from general obligations is not treated in SDL. As deontic logics generally neither capture the provenance of a norm (e.g. its author, specification time, or document position), nor a unique identifier for each norm, conflict resolution in formal logic is in many cases untenable as insufficient information about the policies exists to enable choice amongst them. We can rectify this by associating each identified norm (function or specific obligation instance) with a clause that construes that obligation as having come about – this allows us to **express** conflict by acknowledging and tracing the different sources of norm instances. We refer to the origin of the norm instance in a particular *function* defined by a particular *clause* as that norm instances' **situation**.

As shown earlier (e.g. §5.2, §5.6.1) we situate a clause by adding an isAccordingTo role to each function, obligation, permission, or prohibition, where the participant in this role is a clause identifier (such as Clause C.4) which identifies the utterance that

promulgated this clause. Using an occurrence of being_in, the utterance identifier can then be associated with a document position identifier (e.g. Section 1.1), and a document heading (Supply Agreement Between SkyHi and Steelmans). Alternatively, the clause can be associated with an utterer (e.g. Managing Director) or institution that associates itself with the clause (e.g. Steelmans), and an utterance time and place (e.g. Philadelphia, February 8th 2002).

Organizing rules into documents (contracts) and structuring documents through sections, sub-sections, headings, bullet-points and other document formatting or **utterance labelling**, is an important device that is helpful for conflict resolution and is not considered by the rule engines we surveyed earlier (§2.2). The benefit of such an approach, which makes use of *document- or utterance- marker information attached to provisions*, is that rule groups could be referenced in a manner similar to that found in hard-copy specifications. For instance, it is common for contracts to make document-structure references such as: 'all terms of the contract *dated* 1 April 2001 between Steelmans and SkyHi are null and void', or 'the provisions of *Section 3* shall override (that is, conflicting provisions are void)', or 'subject to the conditions listed *below* ...'. These references are used in order to select provisions (see Guideline 8, page 105). In the retail trade, it is common to label a set of provisions using names such as 'The Meal Deal Offer', or 'Summer Specials', or 'Premium Promotion' and to speak of all provisions attached to those names being null for sales (cases) where other offers apply. All of these references are critically dependent on document structure or markers. Exploiting document label information does not preclude provisions from being stored and interrogated through other access paths as well (e.g. the time the provision was written, or semantic contents of provision); document structuring information merely provides an additional convenient organization and interrogation mechanism. Non-written utterances, may also be marked with their location or context of utterance. For example, all offers spoken during a session of 'without prejudice' negotiation, may be void, or all verbal agreements made in international waters may be null.

A mechanism of labelling conclusions with the section of law from which they were derived was employed by Sergot et al. [SSKK86] in their analysis of the British Nationality Act. That paper recommends that rules take the form:

```
[proposition] on [date] by Section [section number] if [conditions]
```

For example:

```
x acquires British citizenship on 16 March 1987 by Section 11.1 if …
```

Conflict-free specifications are assumed and conflicts are removed by redrafting. For instance, the addition of Section 11.2 which specifies exclusions to the conditions of Section 11.1, would require the restatement of Section 11.1 as:

```
x acquires British citizenship on 16 March 1987 by Section 11.1 if …
and not [x is prevented by Section 11.2 from acquiring British citizenship]
```

We see this as problematic because a prima facie provision becomes altered to an all-things-considered provision, and we are no longer concluding

```
x acquires British citizenship on 16 March 1987 by Section 11.1
```

but rather, effectively,

```
x acquires British citizenship on 16 March 1987 by the law as a whole
```

The provisions of the section are being confused with the provisions of the law as a whole, thereby diluting the usefulness of the section identifier. Our approach is to treat utterances as immutable once uttered and accept that clauses may conflict. Whereas Sergot et al. employ "contraction-and-revision" [HM97], we opt also for "restrained application": rules are left exactly as stated and the norm interpreter chooses one conclusion above another when application of the rules gives contradictory results.

We make the distinction, shown in Figure 16, between **utterances**, which are an *immutable* part of history, and **document labels**. Document labels are attached to utterances. The separation between **utterance provenance** and **document provenance** is critical: as seen in §5.6.5 and §5.6.9, obligations have their origin in particular *utterances* (identified by occurrence or function identifiers). The utterances, in turn, are sourced in various named documents. The utterances that make up a document or document portion (e.g. labelled section or clause) may change over time

as the document is revised. Figure 17 demonstrates the notion of **document history**, giving an example of a document undergoing revision.

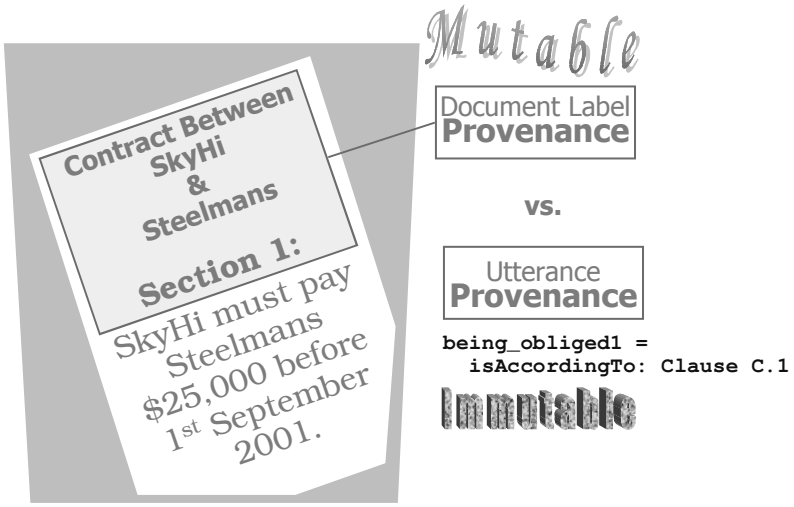


Figure 16: Document vs. utterance provenance

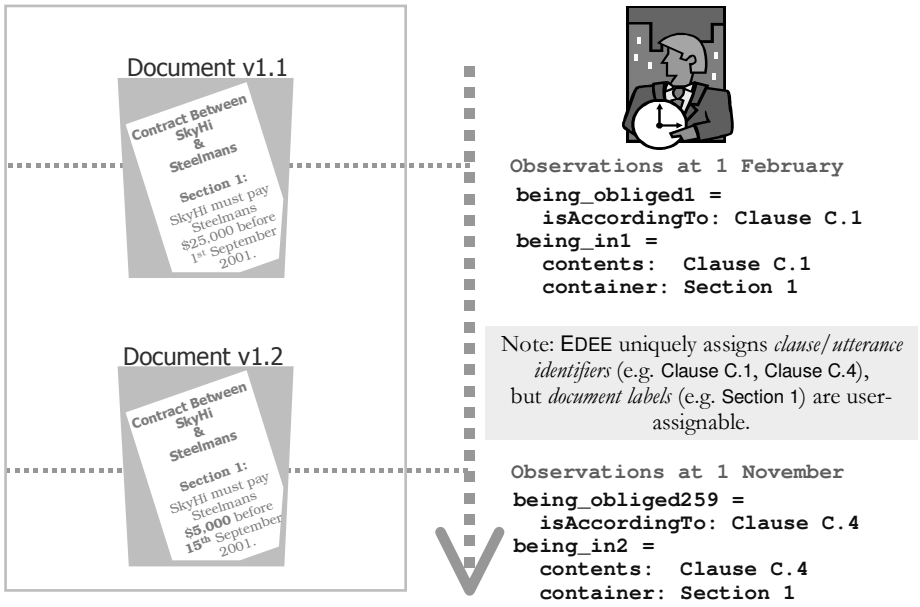


Figure 17: Document history: obligations and document labels

6.2 Detecting Conflict

Since provisions (Chapter 5) are associated with queries that describe sets of occurrences, conflicts may be detected by analytically determining overlap between stored queries. Extending some basic suggestions of deontic logic (§2.6)[24] and policy management systems (§2.3), Table 17 below defines the rules for detecting conflicts[25].

☑ **Requirement 9 (pg 29):** Analytic conflict detection is desirable.

Rule	**Description** (A conflict exists when query overlap shows that …)	Abbreviation
Rule CD1:	An occurrence is in a set of permitted occurrences and a set of prohibited occurrences.	**Permitted and Prohibited**
Rule CD2:	An occurrence is in a set of obliged occurrences and a set of prohibited occurrences (see §6.2.1).	**Obliged but Prohibited**
Rule CD3:	An occurrence is in a set of obligatory occurrences (implying that not performing the action produces a violation), but a permission to refrain from performing the occurrence exists (implying that no violation arises from not performing the action) [Lee88; Mak88]. A special case of this rule is the case of concurrent obligations of different strictness (see §6.2.2 and [AEB2002b]).	**Liable (to violation) but Immune (from violation)**
Rule CD4:	(The function defined in) Clause *x* says an occurrence exists whereas Clause *y* counts zero occurrences fitting that description[26].	**Liable but Immune**

[24] The authors cited here do not make use of a notion of occurrences, but reason at the level of propositions and contradiction between propositions. We restate their suggestions in terms of sets of occurrences.

[25] The level of severity of these conflicts is variable.

[26] This is in fact a generalization of the previous rule, which says that a conflict exists when one clause sees a violation but another counts zero violations.

Rule	**Description** (A conflict exists when query overlap shows that …)	Abbreviation
Rule CD5:	An occurrence is in the domain of a function but is in a set of forbidden occurrences. This principle can be derived from the principle that a power may conflict with a prohibition against exercising that power [Mak86; JS96]. This is because, effectively, functions define powers, through defining what set of occurrences can bring about, according to a certain clause, other occurrences.	**Empowered but Forbidden**
Rule CD6:	An occurrence is in a set of obliged occurrences but is not in the range of a function. This can be derived from the principle that an obligation may conflict with the absence of a power to fulfil that obligation. The latter includes the case where another party has immunity against a certain state of affairs being brought about.	**Obliged but not Empowered**
Rule CD7:	Multiple obligations cannot be fulfilled because of resource limitations. Hansson and Makinson's doctor example (op cit, page 65) is an example of such a conflict. These so-called 'conflicts of priorities' between obligations [MS94] are detectable by estimating reserves and production, and comparing to consumption required to fulfil the identified obligations.	**Obliged but not Able**
Rule CD8:	A function brings about (desirable) occurrences in a range, but the party has insufficient resources to perform occurrences in the domain of the function.	**Empowered but not Able**
Rule CD9:	An occurrence is in a set of permitted occurrences, but the party has insufficient resources to avail themselves of their permission.	**Permitted but not Able**

Table 17: Conflict detection rules

Conflict detection is best illustrated through examples:

6.2.1 Example 1: Obligation conflicts with prohibition

Let us record, in a new and otherwise empty data store, that Steelmans is a supplier of SkyHi, by inserting an occurrence, being_supplier1, as described in Table 3 (page 80). Further, assume that a prohibition against paying more than $10,000 to suppliers is provided by Clause P.3, and stored as illustrated in Table 5 (page 130).

Consider now that SkyHi is obliged, according to Clause C.1, to pay $25,000, before 1 September 2001, to Steelmans for a delivery. Assume that this payment has been contemplated, but not effected – i.e. it has not yet occurred; no occurrence of paying has been added to the data store. As shown in Table 11 (page 143) the obligation is recorded by storing an occurrence, being_obliged1, where the *obliged* (that is, *contemplated*) occurrences are described by the query Query19.

Now, applying rule Rule CD2 (page 174), and comparing the description of the *obliged* occurrences (Query19) to other queries stored in the database, even in the absence of payment data, our coverage-checker finds that, given the existence of being_supplier1, Query19 analytically overlaps with Query10 (§3.3.3). Further, Query10 is a description of *prohibited* occurrences (by virtue of Query10's participation in prohibiting1; Table 5, page 130).

We have thus shown that what is obliged (the description of the obliged occurrences) in this context is covered by what is prohibited (the description of the prohibited occurrences). This indicates a *conflict*. This situation is illustrated graphically in Figure 18.

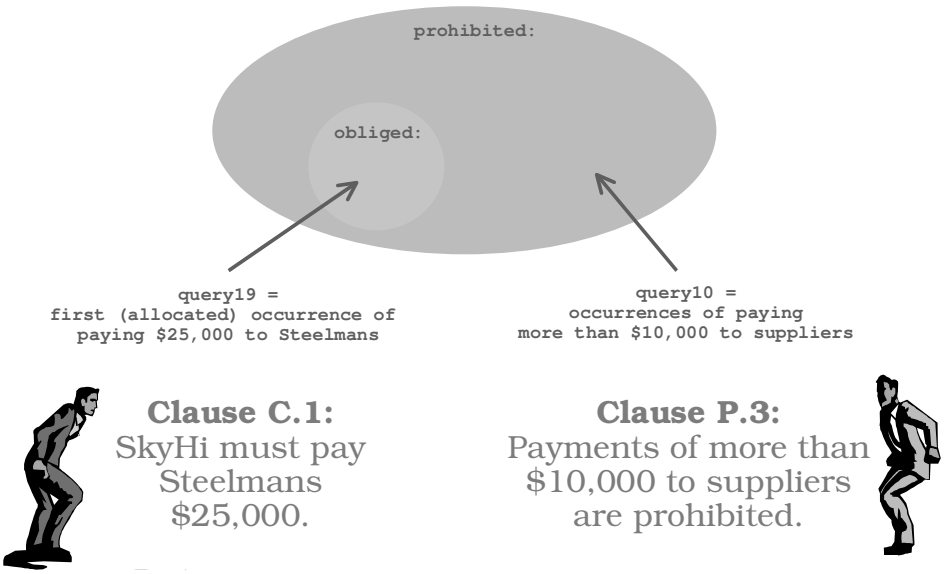


Figure 18: Conflict shown by overlap between obliged and prohibited occurrences

This example demonstrates that EDEE's coverage-detection facility is capable of detecting the case where a prohibition **dynamically comes into conflict** with an obligation: as is evident from §3.3.3 and §3.3.4, the conflict appears when the occurrence, being_supplier1 (indicating that Steelmans is a supplier), is inserted and brings Query10 (from the prohibition) and Query19 (from the obligation) into overlap. Revisiting the covering relations graphs shown earlier in Figure 5 and Figure 6 (page 93), let us now highlight Query19 (the occurrences obliged by being_obliged1) and Query10 (the occurrences prohibited by prohibiting1). Figure 19 shows that, *before* the addition of being_supplier1, there is no conflict between the obligation and the prohibition, since there is no route from Query19 to Query10 in the covering relations graph. Figure 20 shows how the addition of being_supplier1 introduces a route in the graph from Query19 to Query10 and therefore brings about a conflict between the obligation and the prohibition.

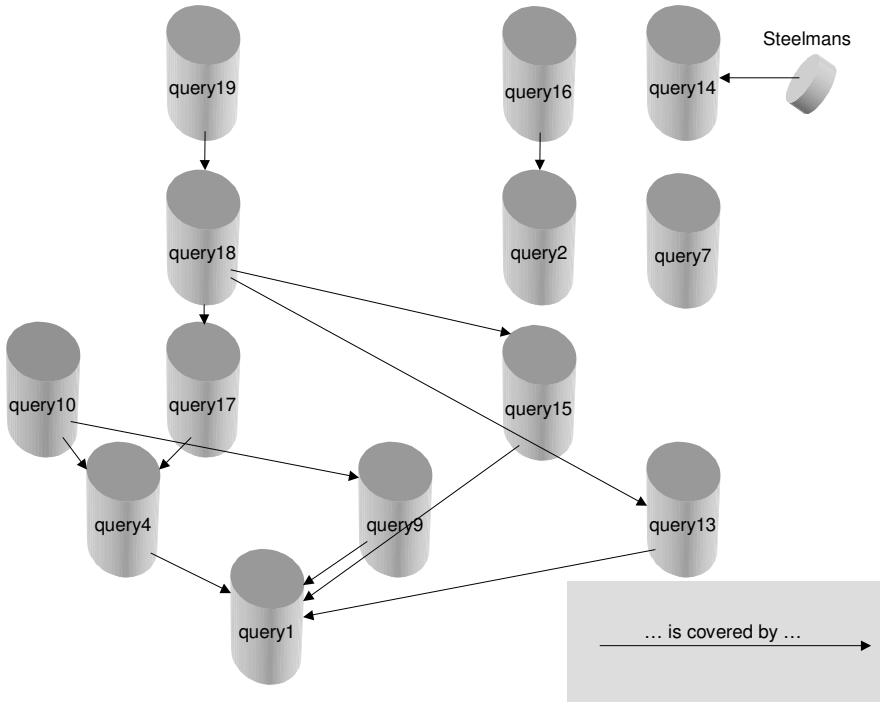


Figure 19: Covering relations graph *before* addition of being_supplier1, highlighting the obliged and prohibited occurrences, with no route connecting them

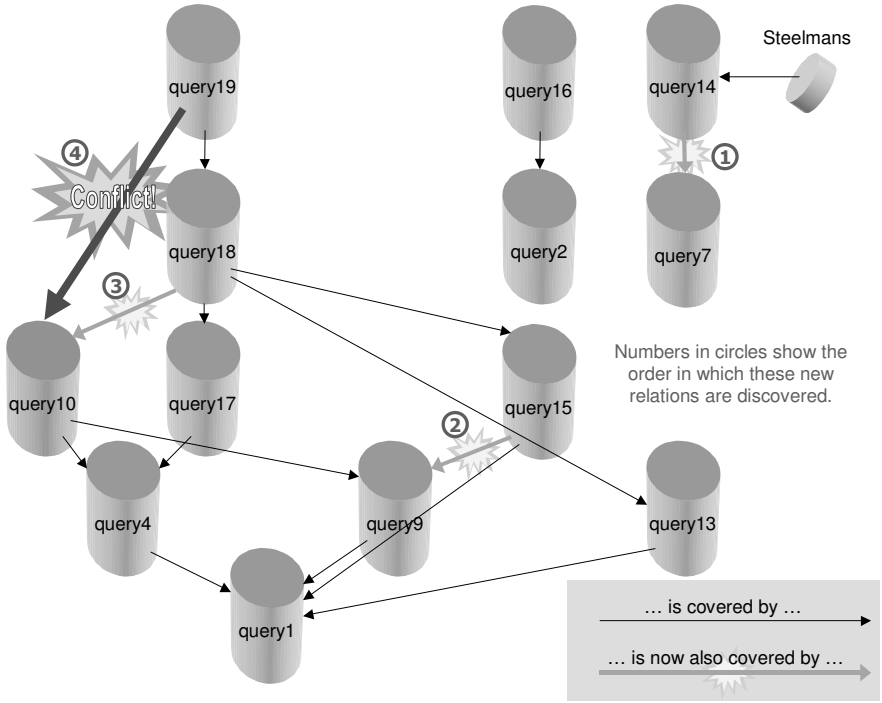


Figure 20: Covering relations graph *after* addition of being_supplier1, highlighting the obliged and prohibited occurrences, the connecting route, and the dynamically-discovered conflict

We can store the conflict as:

```
conflicting1 =
  conflictor:        being_obliged1
  conflictor:        prohibiting1
```

We will see in §6.3.1 how this recorded conflict can be used to show that an agent acted outside their authority, and that, in terms of some commercial laws of agency, the prima facie obligation to pay is therefore not binding.

6.2.2 Example 2: Obligations of different strictness

SkyHi might have a prima facie obligation, according to Clause C.1 (in the Contract Between SkyHi and Steelmans entered into on 1st August 2001 of §1.2) to pay *before 1st September 2001*. But, SkyHi may also have another prima facie obligation, according to Clause C.8 (of their Standard Terms and Conditions) to pay *within 30 days of delivery*. In the case where delivery was made on 15th September 2001, the obligation to pay for delivering1 by 1st September and the obligation to pay for that same delivery by 15th October (30 days from 15 September) are in conflict.

At first sight, it appears that this conflict could be detected by noticing that the queries 'first of (occurrences of paying for delivering1 before 1st September)' and 'first of (occurrences of paying for delivering1 before 15th October)' – which define what is obliged in each case – have a subset-superset relationship. Diagrammatically, Figure 21 below seems an intuitive rendition of the conflict.

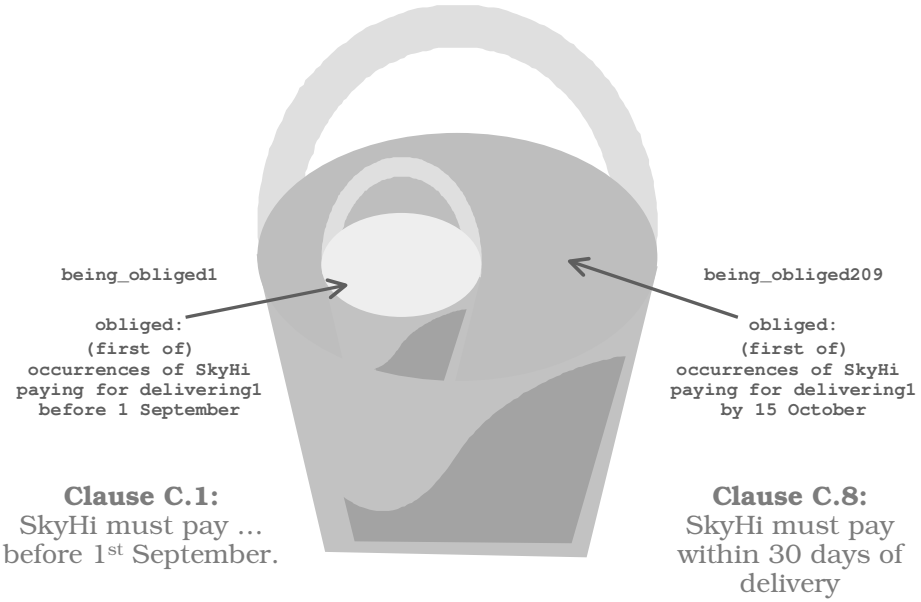


Figure 21: Initial view: conflict between obligations of different strictness

In fact, the view given in Figure 21 above is oversimplified. Obligations of different strictness are more correctly detected through overlap between obligations and the implicit inviolable permissions (§5.4.2) associated with other obligations. All obligations in law seem to confer the following implicit privilege: *the duty-bound party is immune (§5.3.2) from being in violation of the obligation as long as he still has opportunity to fulfil it (and intends to do so).* Taking the obligation in Clause C.8, to deliver within 30 days, this generates an implicit privilege to deliver within those 30 days – that is, non-delivery at any stage before the end of the 30 days does *not* result in violation[27]. By Rule CD3 or Rule CD4 (page 174), this clearly conflicts with Clause C.1 which says that non-delivery by 30th September *does* result in violation.

[27] More formally, there are zero occurrences of violating whose source_occurrence is a counting of zero (0) deliveries, where the counting occurs before the end of the thirty days (15 October).

6.3 Resolving Conflict

In the previous subsection we looked at how conflicts may be detected. We can now pursue conflict resolution. A variety of conflict resolution options are available to us:

- **Rule Revision:** Rules are *revised* to eliminate conflict. This approach is taken by Sergot et al. (op cit, page 172).
- **Case-specific Voidance:** Rules are *amended* to specify which prima facie obligation, permission, or prohibition instances are *void* for a particular case.
- **Acceptance of Violation:** In the absence of revision or amendment, it is *accepted* that fulfilling one provision may *violate* others.

☑
Requirement 14 (pg 32):
Context-specific rule precedence must be supported.

In the presence of conflicting provisions, we may wish to have one or more of the provisions *voided* (see [AB2002b, AEB2002c], §5.6.9), or we may wish simply to guide the decision-maker as to which provisions to *violate*. While we agree that *revisions* to the rule-set may be useful in removing conflict, we see revision alone as insufficient. Supplementation of the rule-set with additional choice principles for case-based reasoning is a necessary and complementary conflict resolution mechanism: here we specify which case-specific obligation, permission, or prohibition instances are *voided*. The rule-set could also be supplemented with choice principles for deciding which norms to *violate* in cases where violations are unavoidable.

The following examples, which follow from our conflict detection examples of §6.2, clarify the conflict resolution process:

6.3.1 Example 1: Resolving a conflict between an obligation and a prohibition

In §6.2.1 above, we detected a conflict between an obligation to pay and a prohibition against paying. We can resolve this conflict by defining a function that specifies which provision's construal (i.e. which instantiation of a norm) is void in the light of conflicting prima facie instances. First, we need to make explicit a number of laws and social conventions to explain the origin of our obligations and the justifications for their voidance:

Powers of Agents: Settled Principles of English Law[28]

…

An occurrence of an agent *promising* something on behalf of their principal brings about an occurrence of their principal *being obliged* to do the promised action. **Clause L.5**

An occurrence of *being aware* of a prohibition that conflicts[29] with an obligation in that contract, brings about an occurrence of the obligation *being void*[30]. That is, an obligation is void in the event that the reliant party knew that the agent did not have the authority to enter into the contract. **Clause L.6**

…

[28] As pointed out by Dr Roderick Munday, Fellow in Law at Peterhouse, Cambridge, and author of *An Outline of the Law of Agency* (1998; 4th ed.; Butterworths), our example case is not covered by any particular Act of English legislation. The rules mentioned here therefore do not originate from any promulgated Act in Britain. Rather, the rules attempt to make explicit some basic settled principles of English case law.

[29] We assume that the law applies when the party is aware of the *provision* that conflicts, even if the party is not aware of the *conflict* of provisions. We take it that the law assumes the party derives the conflict from their awareness of the provision, and even if they do not derive the existence of the conflict, they would have been reasonably expected to have derived it, since the reasoning system is able to derive it automatically itself. 'Reasonable expectation' in law typically means that some individual draws that conclusion; taking the objective EDEE reasoning system as the appointed individual, we might take the fact that EDEE is able to derive the conflict to mean that the party themselves is reasonably expected to know of the conflict.

[30] Technically, we should specify that it is only obligations in contracts entered into *after* the occurrence of *being aware* that are voided. However, for brevity, we omit this complication from our example.

Social Conventions

…

An occurrence of a party *reading* a given clause brings about an occurrence of that party *being aware* of the legal relation mentioned by that clause (that is, the obligation, prohibition, or function associated with that clause). **Clause S.1**

An occurrence of a party P being obliged, in obligation Q, to pay party R, brings about an occurrence of R *being beneficiary* of the obligation Q. **Clause S.2**

…

We posited, in §5.6.1, the existence of being_obliged1 (the obligation of Steelmans to pay), without explanation of its origin in law and circumstance. Clause L.5 explains the source of this obligation: it arose as a result of a particular promise. Let us assume that the promise to pay was made by John, a clerk at SkyHi. The promise may be represented as promising1 as depicted in Table 18 below.

Occurrence	Role	Participant
promising1	promiser	John
	promised	Query19

Query19 (see page 84) = first (occurrence of SkyHi paying Steelmans $25,000 for the delivery, where paying is before 1 September 2001)

Table 18: Representing a promise by an agent on behalf of a principal

We can represent the rule that a promise by an agent binds their principal (Clause L.5) using being_obliged_function5 as shown in Table 19 below.

Occurrence	Role	Participant
being_obliged_function5	domain	Query528
	obliged	Query529
	isAccordingTo	Clause L.5

Query528 = occurrences of promising

Query529 = participants in role [=promised] in |Query528|

Table 19: Representing the rule that a promise by an agent binds their principal

Looking at being_obliged1 in more detail, we would then see its origin and nature as follows:

being_obliged1 =		(legal consequence)
source_rule:	being_obliged_function5	(provenance)
source_occurrence:	promising1	(evidence)
obliged:	Query19 (see Table 18)	

Query19 in being_obliged1 above was obtained by resolving participants in role [=promised] in promising1 – i.e. by resolving the instantiation, for promising1, of the parameterizable query, Query529, in Table 19 above.

Clause L.6, Clause S.1, and Clause S.2 (page 182) would be represented as shown in Table 20, Table 21, and Table 22 respectively.

Occurrence	Role	Participant
being_void_function1	domain	Query530
	voided	Query531
	isAccordingTo	Clause L.6

Query530 = occurrences of being_aware

Informally, Query531 = obligations that are in conflict with provisions the aware party is aware of which are also obligations where the aware party benefits

Formally, Query531 = (occurrences of being_obliged) intersection (participants in role [=conflictor] in occurrences where participants in role [=conflictor] are participants in the role [=aware_of] in |Query530|) intersection (participants in role [=benefit] in occurrences where participant in role [=beneficiary] is participant in role [=aware] in |Query530|)

Table 20: Representing the rule that a prima facie obligation on a principal is *voided* in the case that the reliant party was aware of lack of authority of the agent

Occurrence	Role	Participant
being_aware_function1	domain	Query532
	aware	Query533
	aware_of	Query534
	isAccordingTo	Clause S.1

Query532 = occurrences of reading

Query533 = participants in role [=reader] of |Query532|

Query534 = occurrences where [participant in role [=read] in |Query532|] is in role [=isAccordingTo]

Table 21: Representing the rule that an occurrence of *reading* a clause brings about an occurrence of *being aware* of the legal relation directly mentioned by the clause

Occurrence	Role	Participant
being_beneficiary_function1	domain	Query535
	benefit	|Query535|
	beneficiary	Query536
	isAccordingTo	Clause S.2

Query535 = occurrences of being_obliged

Informally, Query536 = 'payee' in the obligation in |Query535|

Formally, Query536 = results of the query returned by (Value where CriterionType is [=participant] and QueryID is in ((subqueries of the query returned by (participants in role [=obliged] in |Query535|)) intersection (QueryIDs where CriterionType is [=role] and Value is the query ([=payee]))))

(see Figure 3, page 85, for an example of a stored query)

Table 22: Representing the rule that *being payee* in an obligation to pay implies *being beneficiary* of that obligation

Assume that an employee of Steelmans has browsed SkyHi's internal regulations as published on their web-site, and SkyHi's occurrence store had captured the occurrence, reading1, which is an occurrence where Clause P.3 (defining the prohibition against large payments; page 5) was read, and the reader was an agent of Steelmans, who we record simply as Steelmans. That is, assume the occurrence store captures the following occurrence:

reading1 =

read:	Clause P.3
reader:	Steelmans

Now, based on Clause S.2 (Table 22 above), it is easily shown that the existence of the obligation being_obliged1 (§5.6.1) implies the existence of an occurrence of being_beneficiary:

being_beneficiary1 =		(legal consequence)
source_rule:	being_beneficiary_function1	(provenance; see Table 22 above)
source_occurrence:	being_obliged1	(evidence)
benefit:	being_obliged1	
beneficiary:	Steelmans	(payee in being_obliged1)

When the occurrence reading1 is added to the occurrence store, the coverage-checker determines that it is covered by the query Query532 and is therefore in the domain of the function being_aware_function1 (defined in Table 21 above). Thus reading1 brings about the occurrence:

being_aware1 =		(legal consequence)
source_rule:	being_aware_function1	(provenance; see Table 21, p185)
source_occurrence:	reading1	(evidence)
aware:	Steelmans	
aware_of:	prohibiting1	

To explain how we arrived at the values for being_aware1 above: assuming we have reading1 as defined above, and prohibiting1 as defined on page 130, then resolving the query occurrences where [participant in role [=read] in reading1] is in role [=isAccordingTo] (that is, the instantiation, for reading1, of Query534 in Table 21 above) produces prohibiting1. prohibiting1 is therefore in the role aware_of in being_aware1. Similarly, Steelmans in the role aware in being_aware1 was obtained by resolution of participants in role [=reader] of reading1 (that is, the instantiation, for reading1, of Query533 in Table 21 above).

Now, our coverage-checker is able to determine that being_aware1 is covered by Query530 in Table 20 above. Thus, this occurrence of being_aware is within the domain of being_void_function1 and generates the being_void occurrence being_void1. It is being_obliged1 that is in the role voided:

being_void1 =		(legal consequence)
source_rule:	being_void_function1	(provenance; see Table 20, p185)
source_occurrence:	being_aware1	(evidence)
voided:	being_obliged1	

To summarize, taking the facts:

reading1 (see p186) =	Steelmans reads Clause P.3
being_obliged1 (see p184) =	SkyHi (prima facie, according to Clause C.1) being obliged to pay Steelmans $25,000
prohibiting1 (see p130) =	payments greater than $10,000 are prohibited

… we are able to infer, respectively:

being_aware1 (see p187) =	Steelmans being aware of prohibiting1
being_beneficiary1 (see p186) =	Steelmans being beneficiary of being_obliged1
conflicting1 (see p179) =	the prima facie obligation, being_obliged1, conflicts with prohibiting1 (see §6.2.1).

We are therefore able to deduce:

being_void1 (see p187) =	the prima facie obligation, being_obliged1, being void.

… because Steelmans is a beneficiary of being_obliged1, and was aware of prohibiting1 which is conflicting with being_obliged1.

Clearly, being_obliged1, which participates in role voided in an occurrence of being_void, no longer falls into the domain of being_obliged_function4 (page 164), which defined all-things-considered obligations (Clause L.4, page 164). Therefore, a query such as 'what are the obligations, in terms of Clause L.4, that bind SkyHi?' would return no results, and we conclude that, all things considered, SkyHi is not bound to make any payments.

It should be noted here that it is the *case-specific instantiation* (being_obliged1; see page 184) of the obligation policy that is voided, rather than the *policy in general* (being_obliged_function5, page 184). Hence, the general obligation upon principals may still apply to other occurrences of promising, which continue to bring about prima facie, and perhaps all-things-considered, obligations. The distinction between case-specific obligation (similarly, permission and prohibition) instances and the general obligation (similarly, permission or prohibition) policies from which they are derived is at the heart of Hansson and Makinson's [HM97] contrast between "restrained

application" and "revision". In restrained application (typically used by judges in their application of the law), it is the case-specific obligation instance that is voided. In revision (typically used by legislators in their revision of the law), it is the general obligation policy that is voided.

It may be argued at this stage that if we ask the question 'is payment obliged?' we get the response 'yes', since being_obliged1 is stored in our database. However, the 'yes' is actually a qualified 'yes': what we really mean is 'prima facie, according to Clause L.5 (see page 182), yes'. But looking further we see that being_obliged1 is voided, and does not result in an all-things-considered obligation in terms of Clause L.4 as it does not fall into the domain of being_obliged_function4 (page 164).

Note that, just as we have defined what it means to be a current obligation (using being_obliged_function4), we would need to define similar functions to define what it means to be a current permission or prohibition, in order to allow prima facie permission and prohibition instances to be voided as well. For instance, taking the case where SkyHi's internal regulations were not published, and Steelmans was not aware of the absence of authority of their agent, SkyHi may find itself compelled to void (or violate) the prohibition against large payments for this case where it must satisfy the undefeated obligation.

☑

Requirement 13 (pg 32):

Reasoning techniques should emulate legal reasoning. Conflict resolution facilities should allow selection of applicable rules based on recency, specificity, location, authority, or other criteria [GLC99].

6.3.2 Example 2: Resolving a conflict between obligations of different strictness

Relating to the conflict detected in §6.2.2 above, we have two prima facie obligations to pay: the obligation (from the contract) to pay before 1st September and the obligation (based on SkyHi's Standard Terms and Conditions) to pay before 15th October. It is of course possible to fulfil both obligations by, for instance, paying on 30th August, but it may the case that one of the obligations should be voided. Conflict resolution involves selecting which of these conflicting prima facie

obligation instances to void; the remaining obligations (produced as output of being_obliged_function4, which is defined on page 164) are the all-things-considered obligations.

Voiding obligations produced by conflicting clauses is one possible solution to conflict; another, is to merely *accept* that conflict exists and leave both obligations in force. In many cases, it may be that the stricter obligation is intended to be enforced even when the more lenient obligation also applies. Here one obligation does not void the other; rather they exist in conflict, and the company must choose which to violate. If SkyHi pays on 10th October there is then one violation: a violation of their contract with Steelmans. If SkyHi never pays, they have violated both obligations: there is a violation of the contract and a violation of their Standard Terms and Conditions.

☑

Requirement 15 (pg 33): The treatment of obligations – whether to void or violate them in a given case – must be user-definable.

6.4 Time

In the spirit of Sergot et al. (op cit, page 172), we tag each conclusion with a time to indicate the moment at which the conclusion was derived. A conclusion derived at time *t1*, might not be derived at time *t2*. Nonmonotonicity (revision of conclusions) is supported, as partial information is supplemented over time. A continuous query mechanism, which monitors how the results of stored queries change over time still remains applicable here. Occurrences join and leave the domain of functions over time; consequently, the ranges of the functions change over time. Where necessary, we can tag the item produced by the function with the time of its production.

We defined, in §5.6.9, a high-level principle (Clause L.4) that said that something is currently obliged if there is an obligation to do it which has not yet ceased. Now, beginning with an empty database, let us trace our unauthorized-agent scenario (§6.3.1) from its start until the obligation to pay is voided. Figure 22 below gives a commented transcript: % represents a command prompt with user input, > represents a system response to a query, and => represents an output from system derivation.

For brevity, role-players in each identified occurrence are not indicated; the reader may consult the respective occurrence identifiers mentioned earlier in this document (see page references down the right hand margin of Figure 22) for further details on the participants in each identified occurrence and their roles.

Date	Commentary	Session Transcript
1 Jan 2001	A party is only obliged to fulfil obligations that have not ceased (Clause L.4; pg 164)	`% INSERT being_obliged_function4` (pg 164)
	Is SkyHi obliged to pay, according to Clause L.4?	`% SELECT occurrences of being_obliged WHERE isAccordingTo=Clause L.4 AND obliged >= Query19` (pg 84)
	No (no obligations exist yet).	`> 0 results.`
1 Aug 2001	John promises SkyHi will pay Steelmans $25,000	`% INSERT promising1` (pg 183)
	SkyHi is obliged to pay Steelmans $25,000	`=> being_obliged_function5(promising1)` (alias: `being_obliged1`) (pg 184)
	Is SkyHi obliged to pay, according to Clause L.4?	(Same `SELECT` query as above)
	Yes, they are. (`being_obliged1` is in the range of the function `being_obliged_function4`, and the function therefore produces a result.)	`> being_obliged_function4 (being_obliged1)` `> 1 result(s).`
10 Aug 2001	Web log evidence now indicates that Steelmans read Clause P.3 prior to entering into the contract.	`% INSERT reading1` (pg 186)
	Therefore, Steelmans was aware that payments of more than $10,000 were against company policy	`=> being_aware_function1(reading1)` (alias: `being_aware1`) (pg 187)
	…	…
	Therefore, according to Clause L.6 (pg 182), the obligation to pay is voided.	`=> being_void_function1(being_obliged1)` (alias: `being_void1`) (pg 187)
	Is SkyHi obliged to pay, according to Clause L.4 ?	(Same `SELECT` query as above)
	No, they are not.	`> 0 results.`
	Reason: The prima facie obligation is void and has therefore ceased. being_obliged_function4 therefore no longer produces an all-things-considered 'obligation in terms of Clause L.4'.	

Figure 22: Transcript of a session:
adding and querying contracting and workflow occurrences

From the transcript in Figure 22 above, we can see that the question 'is SkyHi obliged to pay, according to Clause L.4?' is answered 'No' on 1st Jan 2001, 'Yes' after they enter into the contract on 1st August, and 'No' again after we discover (on 10th

August) that they were aware that SkyHi's agent was acting outside of his authority. Conclusions are therefore defeasible.

Notice also that, on 15th August 2001, we could ask whether it was the case, according to what we knew on 2nd August 2001 at 14h00, that SkyHi was obliged to pay. We could do this by running the SELECT query from the transcript above, but instructing the inference engine only to use facts and rules dated on or before 2nd August 2001 at 14h00[31]. The query, framed on 15th August but using only information known up until 2nd August 2001 at 14h00, would return `being_obliged_function4(being_obliged1)` meaning that, yes, given what we knew as at 2nd August 2001 at 14h00, SkyHi was obliged to pay.

6.5 Summary

We have posed an important and neglected problem for web-services: ensuring that the business relationship formed when a service is requested conforms with the changing organizational policies of the businesses concerned. Our conflict detection and resolution mechanisms present a novel solution to this problem. Such facilities are an essential requirement in order to maintain the consistency of a dynamically unfolding e-service ecosystem.

In the previous chapter (Chapter 5), we critically reviewed the traditional operator-based construal of obligation and permission in deontic logic and illustrated the usefulness of *identifying* specific obligation, permission, and prohibition instances, which may be generated from norms by functions on occurrences. We showed (§5.6.9) that the identification of separate case-specific instantiations of obligations (similarly, permissions and prohibitions) allows us to speak of the *life-cycle* of those obligation (permission, prohibition) instances, and to specify their *origins in evidence*.

In this chapter, we explained how the supplementation of deontic specifications with additional information to capture the *situation* or *provenance* of a norm (e.g.

[31] Naturally, this discussion requires that we tagged our occurrences with both their *occurrence time* (when they happened) and their *observation time* (when we became aware that they happened).

author, document position, place of utterance, etc.) allows us to *express* (§6.1), *detect* (§6.2), and *resolve* (§6.3) conflicts between opinions of prima facie provisions. Through realistic worked examples, we demonstrated the process of reasoning in the light of conflicting norms, which may be voided or violated. We saw how to reason about the force of provisions over *time* as new information becomes available (§6.4). Our exposition enlightens aspects of the life and times of identified, situated, and conflicting instantiations of norms that are not dealt with in current policy-based systems (§2.3) nor in standard operator-based treatments of norms in deontic logic (§2.6).

In the next chapter, we look at monitoring and enforcement in the prototype implementation.

Chapter 7

Monitoring and Enforcing Provisions

A high-level aim of this research is contract-driven applications. A variety of styles for enforcing and executing contracts are available. This chapter highlights a number of detection, intervention, and prevention strategies that may be employed to find (Section 7.1) or avoid (Section 7.2) violations. Section 7.3 reviews the implementation of our software prototype.

7.1 Provision Monitoring

By contract monitoring we mean that, when workflow occurrences are added to the database, our software sees to it that the necessary contractual consequences are inferred. This usually includes the inference of obligations that effectively drive the system by identifying what it is that capable components ought to perform next. Coordination and mutual understanding are facilitated as components are able to use the database to determine which legally-recognized states of affairs hold: for instance, they may unambiguously determine whether a purchase is valid in terms of a particular provision stored in the database.

Multiple modes of contract provision monitoring are possible. In cases where third parties bring about occurrences, violation may not be avoidable, and *immediate detection* of such violations is required. Often, however, policies are not sufficiently critical to warrant constant monitoring, and *delayed detection* may be acceptable.

7.1.1 Immediate detection

Often, detection of a violation is required immediately after an occurrence is added to the database. The coverage-checking mechanism (§3.3) makes use of covering relations between queries, and partial re-evaluation of dirtied queries. The intention is to determine when the results of stored queries change as a new data item is added. For instance, if a new occurrence is covered by a stored query that describes the occurrences in the prohibited role in a prohibition, this indicates that the occurrence is prohibited under that prohibition.

Take the policy 'payments of more than $10,000 to suppliers are prohibited'. Table 5 of §5.3 represented this by embedding Query10 in a row describing the occurrences prohibited by prohibiting1. Now, assume that this prohibition and its associated query (which describes the prohibited occurrences) are stored in an empty database. Then, assume that we record, in this database, that Steelmans is a supplier of SkyHi, by inserting an occurrence, being_supplier1 (as described in Table 3, page 80) into the data-store. Upon insertion of the rows for being_supplier1 the coverage-checking algorithm is activated. The coverage-checking algorithm determines that the new occurrence, being_supplier1, is not prohibited, since the only query that covers it is [occurrences of [being_supplier]] (see §3.3.2) and this query is not in the prohibited role in any prohibition. Now assume that the payment, paying1, is inserted. The coverage-checking algorithm determines that paying1 is covered by Query10. A quick lookup shows us that Query10 is in the Participant column in a row in the database where prohibiting1 is in the Occurrence column. As paying1 is covered by Query10, which describes the set of prohibited occurrences, paying1 is therefore prohibited. Figure 23 gives a high-level illustration of this process.

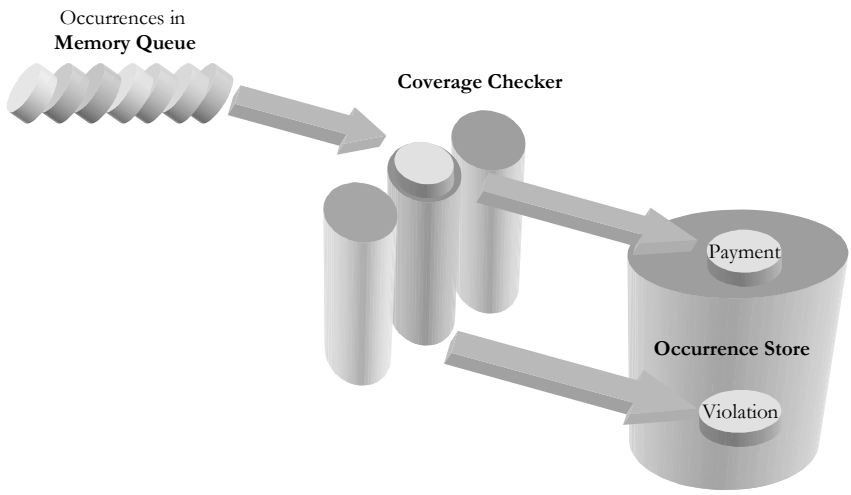


Figure 23: Immediate detection

The computationally-intensive immediate detection algorithm, which checks which queries cover every incoming occurrence, is sometimes not feasible for large data volumes. Another alternative, discussed in the next section, is to buffer occurrences and perform delayed detection.

7.1.2 Delayed detection

The previous section described how the coverage-checking mechanism may be employed for immediate detection of violation: as each occurrence is added, it is checked against stored policies. However, some policies do not require such rigid enforcement. Stationing a full-time security guard at the office supplies cupboard would likely be a sub-optimal allocation of resources. Similarly, checking all occurrences against policies is inappropriate for those occurrences that, individually, are of little consequence. Delays in detecting violation may be traded off against the computational cost of detecting violation immediately. Periodic or aperiodic detection, such as 'at the end of the day', 'on every second Tuesday of the month', or 'at off-peak times of system utilization', may be all that is called for. Thus, to reduce the monitoring burden, configurable evaluation based on the time-criticality of

violation detection should be possible, to allow non-critical policies to be assessed less frequently.

Two primary mechanisms for delayed detection are possible:

1. *Top-down query re-evaluation*:

   The queries associated with selected policies are periodically (or aperiodically) executed. This would be inefficient if many of the queries were unchanged during the interval.

2. *Bottom-up batched occurrence detection*:

   Batch checks of unique identifiers across multiple occurrences may reduce computational cost. In batch checks, unique identifiers added to the database within an interval are fed into the coverage-checking mechanism after a batch of occurrences, rather than after a single occurrence (see Figure 24). The computational saving achievable by this mechanism depends on the frequency of updates of each particular concept identifier. For concept identifiers frequently added to the database the saving would be greater as the concept identifier would be fed into the continuous query mechanism only once for all updates during the interval. For instance, assuming the addition of a thousand occurrences of paying, each with roles payer and payee, an immediate detection mechanism would check which queries mention the role payer one thousand times, and which mention the role payee one thousand times. A batch mechanism would check once which queries mention payer and would check once which queries mention payee. Adelberg, Garcia-Molina and Widom [AGMW97] illustrate, in a stock market application, that a mechanism for batching updates and computing the net effect at the end of a delay window is efficient when data is input in short bursts of similar data.

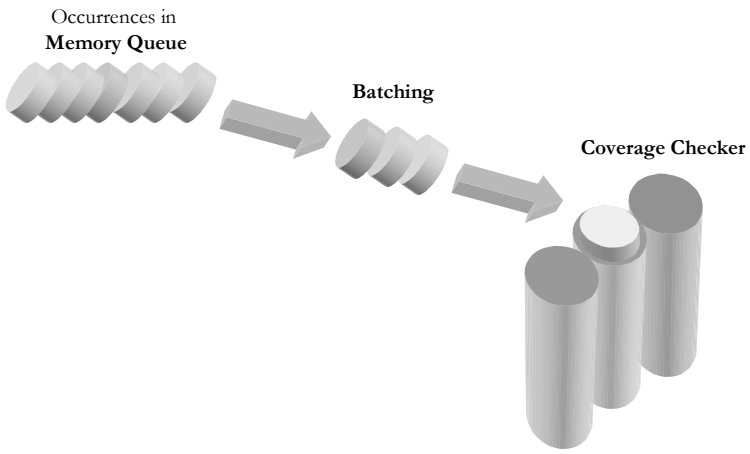


Figure 24: Delayed detection with bottom-up batching

7.2 Performance and Enforcement

Performance and enforcement of a contract may be divided into intervention, prevention by refusal, and prevention by construal. We treat performance and enforcement together since we view enforcement as overlapping with performance: part of enforcement is the performance (typically by a supervisory component) of obligations that arise from violations of a contract. Performance is via *intervention*: we take it that components are diligent in attempting to perform their obligations. We assume that components consult the central active contract database to determine their current obligations and attempt to fulfil them. Enforcement may, however, involve not just intervention, but also prevention. In the case of policies where the consequences of violation are high and control over activity is possible, *prevention by refusal* is appropriate. Refusing a request to execute contemplated action avoids violation. Where control over third-party activity is not possible, *prevention by construal* can be employed. Here, any occurrence not fitting the 'rules of the game' is deemed not to be an occurrence of a certain type in terms of 'the game' (the law), thereby preventing it from having consequence and effect.

7.2.1 Intervention

☑ **Requirement 16 (pg 34):** Asynchronous fulfilment of chosen obligations must be supported as an enforcement style

As we noted earlier (§5.1) our architecture presents a central database where contractual provisions, workflow occurrences, and consequent contractual construals derived from those workflow occurrences are stored. The database stores both the application's specification – that is, the provisions which specify what can, should, and should not happen in various circumstances – and the operational workflow occurrences. This dynamically changing contract database is consulted by the various implementation components to determine what states of affairs hold in terms of the contract, and what occurrences to perform, or refrain from performing, next based on prevailing obligations, permissions, and prohibitions.

Ponder (page 36) provides a policy deployment mechanism whereby language-specific imperative scripts or access control policies are deployed to distributed components. We provide no such mechanism, which in any event limits enforceability of the policies to only the specific platforms supported by the script generators. Instead, we rely on the heterogeneous components to consult the database and determine their obligations, by looking up occurrences of being-obliged that pertain to them; see Figure 25. They can then attempt to fulfil these obligations. Alternatively, we allow facilitator components (e.g. a fulfilment scheduler) to do the consultation and invoke the implementation components.

The process by which diligent components determine and fulfil relevant obligations is called *intervention*, and is one enforcement mechanism. Prevention, in various modes (by *refusal* and by *construal*), is another possible enforcement mechanism.

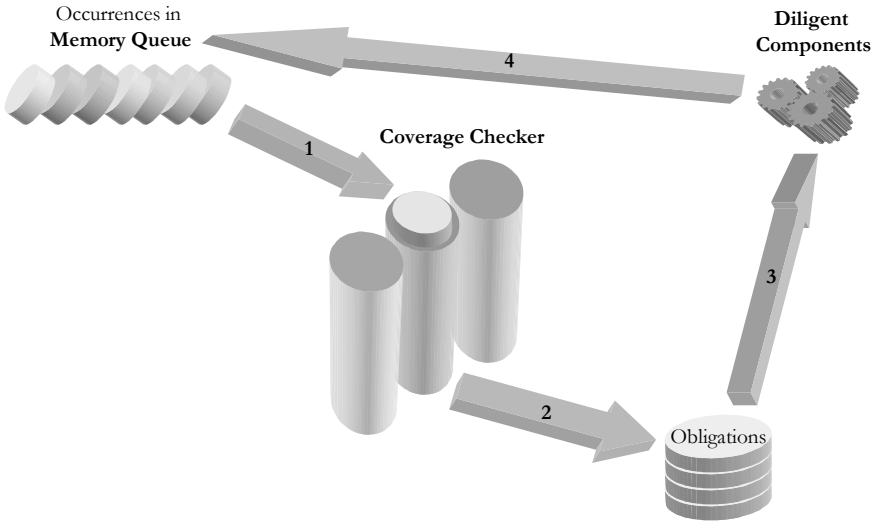


Figure 25: Intervention by diligent components

7.2.2 Prevention by refusal

In prevention by refusal, a decision to block undesirable action, or to refuse requests that might result in prohibited occurrences, avoids violation. Prevention via refusal requires analytic detection of a conflict between a contemplated (e.g. obliged) set of occurrences and a prohibition. We have already seen how such a conflict may be detected (§6.2). If we assume the existence of a choice principle that says the prohibition overrides the obligation in this case, our prevention by refusal mechanism tells us that operations that might bring about an occurrence of paying to fulfil the obligation should not be allowed to execute. That is, if a component is contemplating executing an operation that would bring about such an occurrence, and it requests permission to go ahead from some supervisory or control component, the permission should be refused. We leave it to the implementation component (e.g. firewall, gateway, or access control layer) to see to it that refusal is enforced by blocking an invocation request, for example. The intention of our 'prevention by refusal' mechanism is merely to illustrate that enforcement decisions can be made by determining whether the contemplated (obliged) action is covered by an overriding prohibition.

7.2.3 Prevention by construal

It may be the case that the implementation component is somehow able to bypass controls, flout the refusal, and nevertheless execute a prohibited operation (e.g. a payment), even when they are not legally empowered to bring about a particular legal state of affairs with that name. In this case, a prevention by construal mechanism is necessary as the component has no legal ability to bring about payment-in-some-particular-sense, even though they can actually execute something and call it, in some other sense, a payment. The rogue component may succeed in convincing parties outside the normative system that a payment has occurred but will not be able to mislead parties that consult the contract database to determine whether a particular (legally recognized) payment occurs. In prevention by *construal*, the contract database ensures that, even though a payment has occurred in some sense, it is not construed as such by a particular provision stored in the database and is therefore of no legal consequence.

Forced construals may therefore be used as a preventive mechanism. Here occurrences are regarded as being of a certain type according to a certain clause. It is impossible to coax the occurrence to be of a different type as the database wrapper always appends the clause identifier to the occurrence when it is added to the data store. We saw, in Table 7 of §5.3, how our representation prevents purchases of steel by clerks from being construed as purchases in terms of Clause P.1.

7.3 EDEE Implementation

Many of the concepts described here have been implemented in Java as the E-commerce Application Development and Execution Environment (EDEE). EDEE [AB2001c, AB2001d] is a prototype system for representing and enforcing business policies and contracts. EDEE currently supports immediate detection (§7.1.1), bottom-up batched detection (§7.1.2), and prevention by construal (§7.2.3).

The expressive power of EDEE is that of a semantic network. Networks are stored by reducing a general graph structure to a set of triples (named-and-typed-association, role, participant). Examples of association types are occurrences such as *being-supplier* and *paying* (Table 3, p80), *permitting* (Table 8, p136), *prohibiting*, (Table 5, p130), and so on. The triple store holds the complete representation of the business application, and is updated whenever relevant occurrences take place; such occurrences may include both changes to contractual provisions and transactions such as the payment of a bill or the delivery of an order.

The simple occurrence-role-participant structure of all occurrences enables us to use a broad variety of back-end data stores to uniformly record occurrences, and consequently queries and policies. EDEE is a platform-independent active database wrapper that implements triggers atop arbitrary JDBC-compliant data stores. The ability to uniformly store queries and policies in a vendor-independent manner provides greater platform-independence than proprietary stored-procedures and triggers; furthermore databases that do not support stored procedures and triggers can still be used for active contract storage and enforcement. Contractual provisions can be distributed across heterogeneous remote sites. Simple active databases can be extended with multi-table triggers that can monitor for complex conditions involving multiple abstract data types, rather than simply detecting updates to a single relation as is conventional [PD99].

EDEE incorporates a parser, implemented using the *JFlex* lexical analyser and Princeton University's *CUP Parser Generator for Java*. The parser accepts text containing occurrences and provisions, with embedded queries, from any text stream

(e.g. file read or method invocation) and inserts the relevant tuples into the underlying occurrence store (`TableStore`). When queries are parsed, a check is performed for sub-expressions shared in common with existing stored queries, and semantically identical sub-queries are assigned the same identifier (see *Rule A2.2.3*, page 262). This reduces the number of stored queries and improves coverage checking performance for data sets where the provisions in the database substantially overlap. While policy languages such as Ponder (page 36) provide a clearly defined and readable syntax, EDEE provides simple interfaces for adding contractual provisions and workflow occurrences. Instances of the `Occurrence` class are instantiated, and roles and participants added. `TableStore.addOccurrence(…)` adds the occurrence to the occurrence store in a single transaction. The implementation of a readable contract definition language has been identified as an area for future work (§8.4.2).

Underlying occurrence stores may be a relational database (accessed using the `DBTableStore` class) or in-memory table (accessed using the `MemoryTableStore` class). As both `DBTableStore` and `MemoryTableStore` implement the `TableStore` interface, both persistent and in-memory data are accessible via the same uniform language, EDEEQL, or via method invocations on the `TableStore` interface.

The `resolve(String EdeeQL)` method of EDEE's `TableStore` interface is able to resolve queries specified in EDEEQL against relational or in-memory tables. Relational queries are converted to SQL, and may benefit from optimization by the query optimizer of the underlying database. In-memory queries are resolved directly, without optimization, using Java's `HashMap.get()` operators.

In comparison, traditional relational or object-oriented development approaches would require the use of native programming language code (e.g. Java, C++, or VBScript) to access in-memory data, and either SQL or OQL to access persistent data. In traditional approaches, access to in-memory data is therefore via pre-defined access paths, such as following object-pointers, and advanced ad-hoc queries are not available unless the data is persisted.

EDEE's `CoverageChecker` class is responsible for determining the set of queries that cover a particular item or query. EDEE can express and represent patterns which can be matched against the contents of the triple store. Patterns are implemented by queries; these queries are themselves stored. Contracts are represented by establishing their structure and defining queries to correspond to their provisions. EDEE's `CoverageChecker` component has been supplied to researchers at the University of Aachen, where it is being used for the monitoring of contracts in a business-to-business electronic marketplace [Sta2002].

☑

Requirement 19 (pg 39): Policy environments require a coverage detection service and interface that would allow objects to determine which of a changing set of descriptions they, or other objects, fall under.

Our choice of Java as an implementation approach (over logic programming and theorem proving approaches such as Prolog, PVS, and Isabelle) was driven by a number of factors:

- Java provides platform independence and is standard and pervasively available in industry.
- Query-based reasoning may be more efficient than proof-oriented approaches for large data volumes, as the former takes into account data profiles (predicate selectivity) when executing queries.
- We saw the need to store rules, and not just facts, in a database, rather than merely leaving rules in unmanageable and non-queryable text files. For conflict-detection purposes it is necessary to store queryable contents of the rule (to analytically check for overlap). For conflict-resolution we need to store all rule attributes such as author, creation date, unique identifier of provision and its case-based instantiations. Many Prolog implementations do not support rule storage, and we envisaged that storage of and reasoning with rule attributes and contents, and selecting rules to apply in a circumstance, would require as much effort in Prolog as in Java.
- We preferred a set-based semantics (sets of occurrences) rather than a truth-value-based approach. Query-based reasoning seemed more suited to union, intersection, count, ordering, and other set-based operations on occurrences.

- Data pre-processors, written in efficient imperative languages, were in any event necessary to delay inferencing to cope with spikes in update rates (§7.1.2).

Our system has not yet, however, been tested on actual contracts of any significant size, and scalability beyond toy examples therefore remains to be demonstrated in an extension of our proof-of-concept implementation (see §8.4).

7.4 Summary

This chapter has reviewed the software capabilities of the EDEE prototype environment. We are satisfied that the basic mechanisms are in place to store occurrences (Chapter 3) and provisions (Chapter 5), assess which provisions apply to an occurrence (§3.3.2 and this chapter), and analytically detect a range of static and dynamic conflicts through checking for overlap between stored queries (§3.3.3 and Chapter 6).

In brief, an occurrence-based system performs and enforces contracts by:

- **determining currently applicable obligations**, by finding descriptions of obliged occurrences associated with occurrences of *being-obliged* (§5.6.1) (**diagnosis**)
- **fulfilling its own obligations** by triggering occurrences which it is capable of that fit the descriptions specified in its obligations (§7.2.1) (**liveness**)
- **avoiding the violation of prohibitions** which cover it, by flagging the potential violation to management or by refusing to allow occurrences that fit the descriptions specified in such prohibitions (§7.2.2 and §7.2.3) (**safety**)

- **monitoring fulfilment** of obligations and violation of prohibitions by users, through the determination of covering-queries and creation of instances of violation occurrences (§5.3.1, §5.6.2) (**detection**)
- **bringing about** new occurrences appropriately according to defined norms, thereby causing certain types of occurrences in line with powers (§5.5) and suppressing certain types of occurrences (§5.3.2) (**prevention / immunity**)
- **fulfilling any secondary obligations** (§5.6.2) arising from unavoidable violations (**correction / cure**)

The next chapter evaluates the EDEE prototype environment and the proposals contained in this dissertation.

Chapter 8

Analysis

This chapter provides a *qualitative* synopsis of the features of our approach, and discusses its merits (§8.1) and shortfalls (§8.2). A *quantitative* evaluation of EDEE's performance is also provided (§8.3). Finally, we highlight directions for future research (§8.4).

8.1 Strengths

The major strengths of the approach we propose can be categorized into those that pertain to the occurrence-based *representation* and those that pertain to the *implementation*.

A strength of the EDEE system is its semantic power. The generic store used to represent business applications models a semantic network directly as a set of triples. An occurrence-based representation provides a number of benefits over conventional models: capacity to allow associational queries (§3.1), enhanced schema and code stability through reification of attributes (§8.1.1), a fine-grained dynamic classification facility (§8.1.2), inbuilt support for temporal data and histories (§8.1.3), both object-pattern and pattern-pattern matching (§8.1.4), ability to cater for variable-attribute entities (§8.1.5), and expressive interface advertisements (§8.1.6).

Strengths of the proposed implementation environment include a database independent active wrapper (§8.1.7), multi-table triggers (§8.1.8), and exploitation of query optimization technology (§8.1.9).

8.1.1 Enhanced schema and code stability by reifying attributes

Embley et al. [ELW94] argue that attributes should be eliminated from data models as they introduce unnecessary rigidity, and recommend instead the use of 'relationship sets'. Empirical evidence from Goldstein and Storey in [ELW94] reveals that analysts often mistakenly encode relationships as attributes, making it impossible to state further properties of the attribute without reifying it to a first class object. Halpin [Hal98] explains that schemas often evolve as assumptions about the cardinality of relationships are invalidated. For instance, the assumption that `Manager` is an attribute of `Employee` may be invalidated if the company introduces a matrix-management organizational structure with multiple managers per employee. Such an alteration would require changes to the schema, and rewriting of all SQL queries – potentially distributed across vast numbers of source code files – that refer to the original schema. For example, a change in cardinality relationship between employees and managers requires that the query:

```
SELECT manager_ID FROM Employee WHERE Employee.salary > 10000
```

becomes

```
SELECT manager_id FROM EmployeeManager, Employee
WHERE EmployeeManager.employee_ID = Employee.employee_ID
AND Employee.salary > 10000.
```

Halpin's approach is to define a stable high-level language, Conceptual Queries (**ConQuer**), and to use an algorithm, **RMAP**, to convert the high-level language to the necessary SQL statements as the underlying schema changes.

In our occurrence-centric approach, cardinality constraints do not impact upon the schema: converting from *One* manager manages each employee to *Many* managers manage each Employee involves merely the introduction of further occurrences of

managing, and the alteration of an integrity constraint that checks the number of managers per employee. The absence of a need to alter the schema, or dependent queries, as cardinality assumptions change, indicates that the occurrence-centric schema, and programming code accessing the schema, may enjoy greater stability than traditional approaches in this respect.

8.1.2 Fine-grained dynamic classification facility

Fowler [Fow98] explains that major object-oriented programming languages support only single, static classification, which is unsuitable for the modelling of dynamic roles. Static classification means that an object cannot change its type. In standard object-oriented modelling, `Employee` and `Manager` are typically modelled as sub-classes of `Person`. Discovering that an `Employee` can become a `Manager` requires a tedious object transformation to be coded, frequently resulting in loss of historical information as to the previous status as `Manager`. Even more troublesome is the fact that a person can simultaneously be both an `Employee` and a `Manager`. The creation of a `ManagerEmployee` class may be required to capture this special case, as such a hybrid cannot simply inherit behaviour from both `Employee` and `Manager`. Standard object-oriented models cater poorly for cases where objects play multiple and/or changing roles.

Kappel et al. [KRR2000] echo Fowler's criticism, explaining that traditional class-based object-oriented systems are characterized by an immutable linkage between an object and its class, making it difficult to change the behaviour of an object over time. Any changes would require the creation of a new instance and the transformation of attribute values from instances in the old class to those in the new one. Role classes are proposed as a mechanism for allowing run-time object evolution, enabling objects to play multiple roles simultaneously or over time.

The occurrence-centric approach caters naturally for roles: it is straightforward for a person to participate both in occurrences of being_employed (thereby, being an employee) and of managing (thereby, being a manager). While object-oriented development requires a manual disentanglement of conflicting responsibilities, an

occurrence-centric approach allows automated detection of many conflicts between provisions, which can be flagged for resolution.

To give a concrete example in the object-oriented paradigm, `TenuredEmployee` would typically be a class, and object instance migration would occur from `Employee` to `TenuredEmployee`, perhaps using a periodic batch run or an ongoing daemon process. In contrast, in the occurrence-based approach the individual would participate in an occurrence of being_tenured. Type determination is *a posteriori* as classification and consequent behaviour determination can occur at run-time. The rules, such as pension and leave entitlement provisions, applying to tenured employees, may be different to – and may conflict with – those defined for non-tenured employees. Conflict-detection and determination of the applicable rules can be system-assisted, rather than left to the programmer who would need, in an object-oriented environment, to manually sift through `Employee`, `TenuredEmployee`, and other classes looking for conflicting rules.

Object-oriented classes provide only static, hard-wired ontologies, whereas set descriptions provide dynamic classification facilities. In our approach, we can refer to all items fitting a description. The occurrence-centric model described in this thesis provides powerful knowledge representation capabilities through its ability to specify sets (classes) using queries rather than static class hierarchies. Using intensional queries, rather than extensional sets, improves maintainability, by shielding policies from changes in group membership [Dam2002].

8.1.3 Inbuilt support for temporal data and histories

Continuing the example of employees and managers from §8.1.1 and §8.1.2, it is clear that employees may have multiple managers over time. Encoding `Manager` as an attribute of `Employee` would be a potentially poor design, as this would preclude the firm from storing the history of managers for each employee. Storing this information in a textual log would not suffice, as such a representation is not easily queryable. Recording multiple occurrences of managing, each uniquely identified, and

with distinct participants in the roles manager and managed, is a natural approach to maintaining historical information.

8.1.4 Pattern-pattern matching for conflict detection

RETE (page 26) and TREAT (page 27) are capable of matching constant valued objects to patterns, but are unable to infer the relationships between patterns. For instance, RETE and TREAT are unable to determine that `x <= 4` covers `x < 3`, nor that `A ∪ B` covers `B ∩ C`. EDEE's coverage-checker supports analytical determination of superset, subset, overlap, and disjointedness relationships between dynamically added queries. The ability to determine which policies speak of overlapping sets of occurrences or individuals provides a powerful mechanism for detecting and resolving conflicts.

To our knowledge, our mechanism is the first continuous query or publish-subscribe mechanism to be developed for deontic applications. Previous approaches to checking for permission and prohibition applicability, and for obligation applicability and fulfilment, have been based on Petri Nets or Finite State Machines (§2.5.7). Petri Nets and Finite State Machines do not maintain an event history, require manual re-derivation when new provisions are added, and (as shown in §5.6.7) deal with single but not several, separable obligations.

8.1.5 Ability to cater for variable-attribute entities

Mass customisation, one-to-one marketing, and Customer Relationship Management share a common demand for personalization data, which records the rapidly changing attributes of customers and website content. A company's demands for demographic data, stock-item details, and other facts and beliefs cannot be fixed at application design time. Companies frequently decide to collect and match new information, and find it necessary to extend their database representations of `Customer`, `Product`, or other types with new attributes. Traditional relational schemas are ill-suited to variable attribute storage; sparsely populated tables require repeated renormalization to eliminate null-valued fields. Mainstream content management

solutions, such as **Microsoft Site Server** [Mic2001a], therefore employ a combination of Lightweight Directory Access Protocol (LDAP), HTML `<meta>` tags, XML tags, and indexing services to tag individual users and content with a subset of attributes defined as pertinent to their types. XML databases such as **Ipedo's XML Database** [Ipe2001] and **Software AG's Tamino** [Sof2001] are targeted at storing content where instances of single types may have varying attributes, and different types may share similar attributes. An occurrence-centric representation shares this suitability to sparse data, allowing entities to participate in any number of occurrences subject, of course, to the defined provisions.

8.1.6 Expressive interface advertisements

Simple interface advertisements in conventional trader services employ Interface Definition Languages such as the Object Management Group's IDL [ISO99b]. These interface definitions advertise only single, independent methods. The protocol for accessing the methods is implicit and there is no indication of commitments that come into being as a result of each service invocation. Our paradigm rectifies this with explicit representation of the deontic effects of occurrences brought about by invocations.

8.1.7 Database independent active wrapper

Traditional Event-Condition-Action rule languages (page 15) are implemented as proprietary, database dependent triggers (often based on SQL or OQL) above specific DBMSs and therefore cannot be used as active wrappers for arbitrary relational stores, as is the case with EDEE. Here, and in [AB2001c, AB2001d, AB2002a, AB2002b], we described an active database wrapper capable of running atop any JDBC- and SQL-92-compliant relational datastore[32]. Our architecture therefore provides a platform-independent triggering mechanism.

[32] The database must allow nested `SELECT` queries. **mySQL 4.0**, for instance, while JDBC-compliant, does not yet have facilities for resolving nested `SELECT`s and therefore will not support EDEE.

8.1.8 Multi-table triggers

Unlike regular SQL triggers (page 15), which must be associated with a single table (relation), we are able to define triggers that pertain to multiple relations (occurrences).

8.1.9 Exploitation of query optimization technology

Fixed inferencing strategies, such as those employed by vanilla versions of Prolog, use a search strategy that remains unchanged despite changes in data-set characteristics. This works well for small, in memory data stores but becomes untenable when data volumes grow to thousands of records. An inferencing approach based on database query execution may benefit from indexing, clustering, meta-information, association rules, and other techniques employed by database query optimizers.

8.2 Weaknesses

The weaknesses of the approach adopted can also be categorized into those that pertain to the representation, and those that pertain to the implementation. The main weaknesses of the proposed representation are increased storage space requirements (§8.2.1), and inefficiency caused by frequent, deep graph traversal (§8.2.2). A weakness of the current implementation is its performance (§8.3).

8.2.1 Storage space inefficiency

Repetition of occurrence-, role-, and participant-identifiers in the current schema means that data consumes more space than is strictly necessary. This is partially a product of having attempted to fit our occurrence-centric schema to the constraints of a relational data model. Compression or clustering of data may yield significant reductions in space usage.

8.2.2 Inefficiency of graph traversal

The storage of a semantic network in relational form is sub-optimal, as most simple lookups require multiple queries. For instance, determination of SkyHi's supplier requires two queries: first the occurrences of being_supplier which have SkyHi as supplied must be found, and then the participants in the role supplier must be determined. For transitive data, such as stored occurrences of Query1 (properly) covering Query4, and Query4 (properly) covering Query10, the data structure is particularly inept: determining which queries cover Query10 using these stored occurrences requires first the determination of the occurrences of covering with Query10 as covered, then the determination of the coverers in these coverings, and so on. For optimization purposes, therefore, we have employed an `EdeeCoverer` table with `Covered` and `Coverer` columns, to record 'properly covers' relations (see pages 89 and 95). However, this is still a very inefficient manner of storing these irreflexive, asymmetric, transitive relations.

8.2.3 Slower performance of generic database wrapper

The ability of the EDEE active wrapper to run atop arbitrary relational data stores brings with it performance disadvantages. A major reason for this is the additional overhead of repeatedly parsing queries through JDBC, which introduces an additional processing layer atop the database.

8.3 Performance

EDEE's performance can be reviewed from both a theoretical (§8.3.1) and a practical (§8.3.2) perspective. Direct comparative analysis against other systems (§8.3.3) is difficult owing to the novelty of EDEE's execution paradigm.

8.3.1 Theoretical complexity analysis

Complexity analysis of the coverage determination algorithm shows a polynomial worst-case space complexity of order $O(n^2)$ where n is the number of unique identifiers (including occurrence, role, participant, query, query-criterion, and query-

criterion-value identifiers) in the database. Specifically, the transitive covering relations between concepts and queries, and between queries themselves, is a partial ordering – a directed, acyclic graph – that can be stored (inefficiently) as a matrix in tabular form. For example, to store the transitive covering relation Query10 ⊂ Query4 ⊂ Query1 (from the example in §3.3.4) as tuples that capture the covering relations directly, we could use tuples in the form `<covered, coverer>` to arrive at the 3 tuples: `<query10, query4>`, `<query10, query1>`, `<query4, query1>`. It is easily shown that the number of such tuples required to store the transitive covering relationships between n queries is $(n \times (n-1)) \div 2$, giving a space complexity of $O(n^2)$ for the storage of these covering relations.

The time complexity of the algorithm is also polynomial, with the number of *overlaps* between provisions being more critical to speed than the number of provisions or occurrences *per se*.

8.3.2 Practical experiments

The goal of our experiments was to validate whether our algorithms could, in practice, scale to large contracts on long-running systems.

Parameters

In our experiments, we were able to control the following parameters:

(1) the number of operational workflow occurrences (see §3.1) in the history

(2) the number of legal provisions (see Chapter 5) against which occurrences were to be assessed. Each provision embedded a query composed of approximately 10 nested subqueries (see §3.2).

(3) the size of the batches in which provisions and occurrences were coverage-checked (see §7.1.1 and §7.1.2).

Experimental Setup: Hardware and Software

We undertook multiple runs on diverse platforms. Our algorithm was implemented in Java, and compiled to byte-code, for interpretation on Sun Java Virtual Machines. Table 23 provides the detailed hardware and software specifications of the machines used for our tests.

Machine Name	Operating System	Database	Java	CPU(s)	Memory
teme	Microsoft Windows 2000 Professional	Microsoft Access 2000	1.3.0	800 Mhz AMD Athlon	256 MB
citadel	Microsoft Windows XP Professional	Microsoft Access 2002	1.4.0_01	500 Mhz Pentium III	256 MB
jetset	Microsoft Windows 2000	Microsoft Access 2000	1.3.1	500 Mhz Pentium III	256 MB
All Windows platforms employed Sun's JDBC-ODBC driver, included with their respective Java distributions.					
flute	Red Hat Linux 7.2	PostgreSQL 7.2.1	1.4.0_01	1.4 Ghz AMD Athlon	512 MB
hot-spare-00 (elbe)	Red Hat Linux 7.1	PostgreSQL 7.0.3	1.4.0_01	2 x 1.4 Ghz AMD Athlon	2.5 GB
hot-spare-01 (nidd)	Red Hat Linux 7.1	PostgreSQL 7.0.3	1.4.0_01	2 x 1.4 Ghz AMD Athlon	512 MB
hot-spare-02 (loire)	Red Hat Linux 7.2	PostgreSQL 7.2.1	1.4.0_01	2 x 1.4 Ghz AMD Athlon	882 MB
hot-spare-03 (lyd)	Red Hat Linux 7.1	PostgreSQL 7.0.3	1.4.0_01	1.4 Ghz AMD Athlon	878 MB
gargantubrain	Red Hat Linux 7.1 (bastardised); Linux 2.4.9-18smp kernel	PostgreSQL 7.1.3-3	1.4.1beta	4 x 800 Mhz Intel Itanium (IA-64)	16 GB
All Unix platforms employed the Postgres JDBC driver that is included with the PostgreSQL 7.1 distribution.					

Table 23: Software and hardware specifications of machines used for experiments

Tables and indices in each of the database were set up using a common script, written in Java, with the only variance being that Microsoft Access required the use of the `Double` data-type in the `EdeeNumber` table, whereas Postgres expected the `Float` data-type to be used.

Methodology

We made use of a biased quasi-random provision, occurrence, and participant generator to generate provisions with a high probability of conflict, so as to exercise our conflict detector (see §3.3, and Chapter 6) quite strenuously. We controlled the generation of occurrence types, participant types, and provision types in order to achieve this. The randomizer ensured that:

1. approximately 50% of provisions were obligations, and 50% were prohibitions. Obligations were generated following the general schema of the obligation, being_obliged1, and the prohibition, prohibiting1, from §6.2.1. That is, obligations were "to (some occurrence) (some amount) to (some participant)" instead of simply "to pay $25,000 to Steelmans". Similarly, prohibitions were "against (more-than/less-than) (some amount) (some role) to (some role)", which is more varied than simply "against more-than $10,000 paid to a supplier". Roughly half of the prohibitions (that is, 25% of provisions) were generated in order to produce what might be termed 'dead-end garden paths', where most child sub-queries overlapped but one was disjoint and therefore guaranteed that the parent queries did not overlap.
2. approximately 20% of occurrences were one of 2 types (modelled on being_supplier and paying of §3.1), 20% of occurrences were one of 4 types, a further 20% of occurrence were one of 4 additional types, another 20% of occurrences were also one of 4 further types, and the final 20% of occurrences were one of a million types.
3. roughly half of occurrences had two roles (e.g. *occurrence_type*-er and *occurrence_type*-ed, as in suppli-er, suppli-ed), and the remaining half had three roles (e.g. *occurrence_type*-er, *occurrence_type*-ee, and *occurrence_type*-ent, as in pay-er, pay-ee, paym-ent).
4. approximately 20% of participants were one of 2 individuals (modelled on SkyHi and Steelmans of §3.1), 20% of participants were one of 4 individuals, a

further 20% of participants were one of 4 additional individuals, another 20% of participants were also one of 4 further individuals, and the final 20% of participants were one of a million individuals.

5. approximately 50% of comparison queries were more-than and 50% were less-than.
6. numbers (amounts) were arbitrarily chosen floating point numbers, between 0 and 1,000,000.

This ensured an interesting mix of provisions, and a good spread of occurrence and participant types, with some common types predominating, as would be expected in real business scenarios.

All experimental runs were generated with the same parameters. The same seed (Seed A = 1029353288745) was used to propagate the random generator in all cases, aside from the tests on the machines citadel and jetset where a different seed (Seed B = 1029273107527) was used for variety.

Figure 26 and Table 24 below show the number of unique identifiers stored in the database for various numbers of provisions and occurrences in all tests that used Seed A. As expected, the number of unique identifiers increases linearly with respect to the total number of provisions and occurrences. The number of unique identifiers (n) is important as the overall space utilization is polynomial, specifically n^2, with respect to this (see §8.3.1).

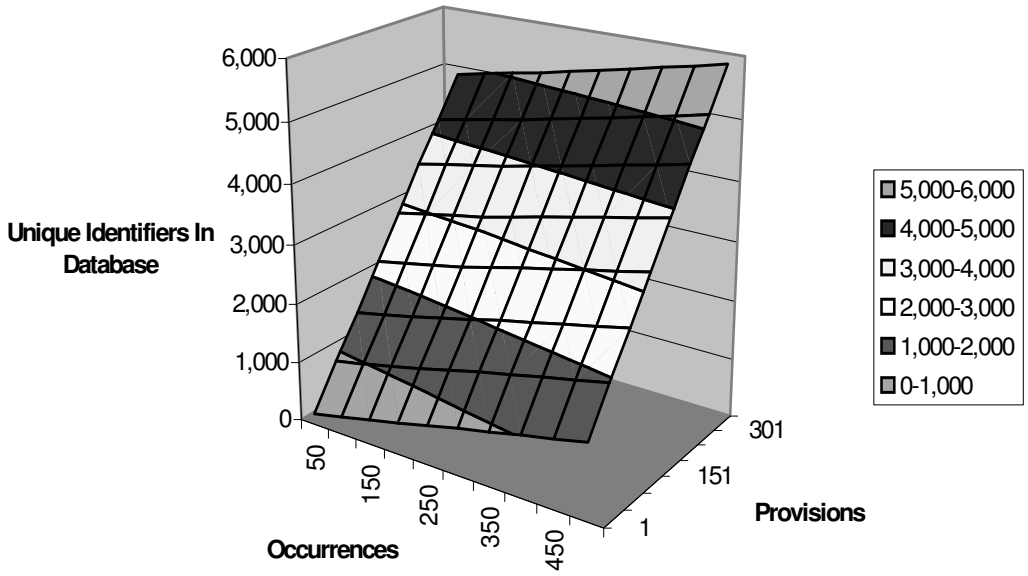


Figure 26: Number of unique identifiers (*n*), as number of provisions and occurrences vary

		Number of Occurrences									
		50	100	150	200	250	300	350	400	450	500
Number of Provisions	1	190	302	418	538	663	794	907	1,029	1,141	1,263
	51	876	982	1,098	1,218	1,343	1,474	1,587	1,709	1,821	1,943
	101	1,508	1,612	1,723	1,843	1,968	2,099	2,212	2,334	2,446	2,568
	151	2,200	2,304	2,403	2,506	2,631	2,762	2,875	2,997	3,109	3,231
	201	2,869	2,973	3,072	3,167	3,287	3,418	3,531	3,653	3,765	3,887
	251	3,559	3,663	3,762	3,857	3,970	4,085	4,198	4,320	4,432	4,554
	301	4,195	4,299	4,398	4,493	4,606	4,716	4,822	4,944	5,056	5,178
	351	4,855	4,959	5,058	5,153	5,266	5,376	5,482	5,602	5,713	5,835

Table 24: Number of unique identifiers (*n*), as number of provisions and occurrences vary

In each experiment, we first inserted the desired number of provisions, and then inserted the requisite number of workflow occurrences. The time of coverage-checking depended on the batch-size: for a batch-size of 1, coverage-checking was immediate, whereas for a batch-size of 50, coverage-checking was delayed until a group of 50 provisions or occurrences had been added (see §7.1.1 and §7.1.2).

Raw Results

In all cases, we recorded:

Time

1. the time taken to insert and coverage-check *provisions* in seconds. Figure 27 and Table 25 below compare the *total* time to input provisions across various machines. The trend-lines overlaid on the data points in Figure 27 are consistent with the polynomial time complexity anticipated in §8.3.1. Figure 28 and Table 26 give the *average* time in seconds per provision. All results here pertain to a batch-size of 1.
2. the time taken to insert and coverage-check *occurrences*, in seconds. Figure 29 (and Table 27) shows the *total* time to input occurrences, as the number of provisions varies, for the best-performing machine, teme. Figure 30 (which is based on Table 27 and Table 28) shows the total time to insert occurrences with 251 stored provisions, comparing batch-size 1 to batch-size 50. Figure 31 (and Table 29) shows the *average* time to input occurrences, as the number of provisions varies, for the best-performing machine, teme. Figure 32 (which is based on Table 29 and Table 30) shows the average time to insert occurrences with 251 stored provisions, comparing batch-size 1 to batch-size 50.

Space

3. the space used by the coverage-checker for *provisions*, calculated as the number of rows added to the `EdeeCoverer` table. Figure 33 and Table 31 show the *total* space utilization for provisions, for all experiments using Seed A. Figure 34 and Table 32 show the *average* space utilization per provision, for varying numbers of provisions.
4. the space used by the coverage-checker for *occurrences*, calculated as the number of rows added to the `EdeeCoverer` table. Figure 35 and Table 33 shows the *total* space utilization for occurrences, for all experiments using Seed A. Note that these graphs show *actual* performance figures, and that the *theoretical* worst-case figures can be computed using the square of the number of unique concepts in the database (see §8.3.1, and Figure 26 on page 221). Though not depicted in the graph, we also periodically recorded size-on-disk for the Microsoft Access databases and found: 0.25MB for an empty database, 0.5MB for 10 provisions and 10 occurrences, and 95MB for 351 provisions and 200 occurrences.

Conflicts

5. the *conflicts* detected between individual obligations and prohibitions. Figure 36 shows the total conflicts detected, for all experiments using Seed A, as the number of provisions and occurrences vary. Figure 37 (which draws its data from Table 34 and Table 35) shows the total conflicts detected, comparing batch-size 1 to batch-size 50. For Figure 37, 251 stored provisions were created, and occurrences were then added.

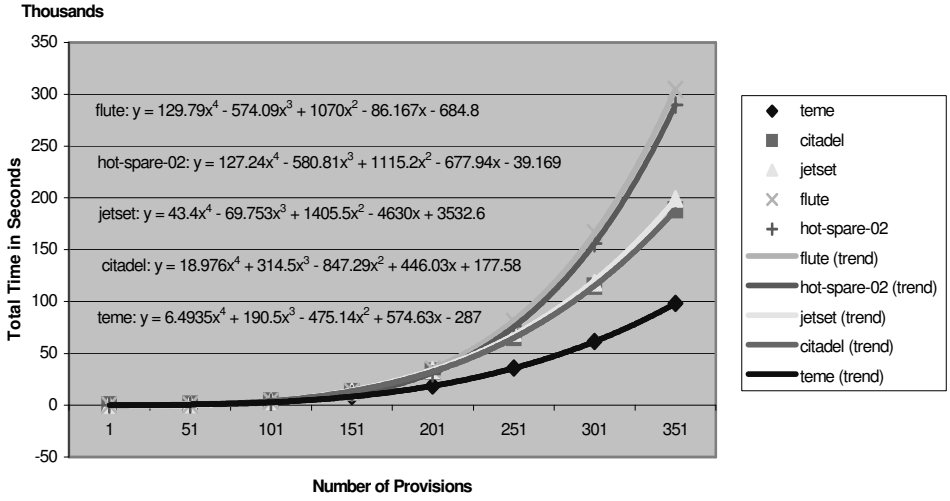


Figure 27: Total time, in seconds, to insert and coverage-check provisions, for best performing installations, with trend-lines fitted

		Machine Name								
		teme	citadel	jetset	flute	hot-spare-00	hot-spare-01	hot-spare-02	hot-spare-03	gargantubrain
Number of Provisions	1	2.5	4.1	2.8	2.3	1.8	1.8	1.0	2.0	2.5
	51	613.0	794.3	768.7	493.2	1,072.3	1,053.3	251.7	1,066.7	5,978.7
	101	2,806.4	3,872.1	3,885.1	3,792.3	38,436.2	38,275.1	2,712.9	38,402.6	235,319.5
	151	8,275.3	12,908.7	13,042.7	13,159.4	330,584.8	328,217.0	10,946.2	328,737.1	
	201	18,546.9	32,606.8	33,748.6	34,852.2			30,414.0		
	251	35,681.3	65,416.0	69,879.0	80,641.8			76,281.1		
	301	61,343.2	114,650.3	118,250.7	166,717.8			155,840.9		
	351	98,047.5	188,430.7	199,033.4	304,569.4			289,757.6		
		Blank cells appear because tests on poorly-performing configurations were manually terminated at an early stage.								

Table 25: Total time, in seconds, to insert and coverage-check provisions, comparing various installations

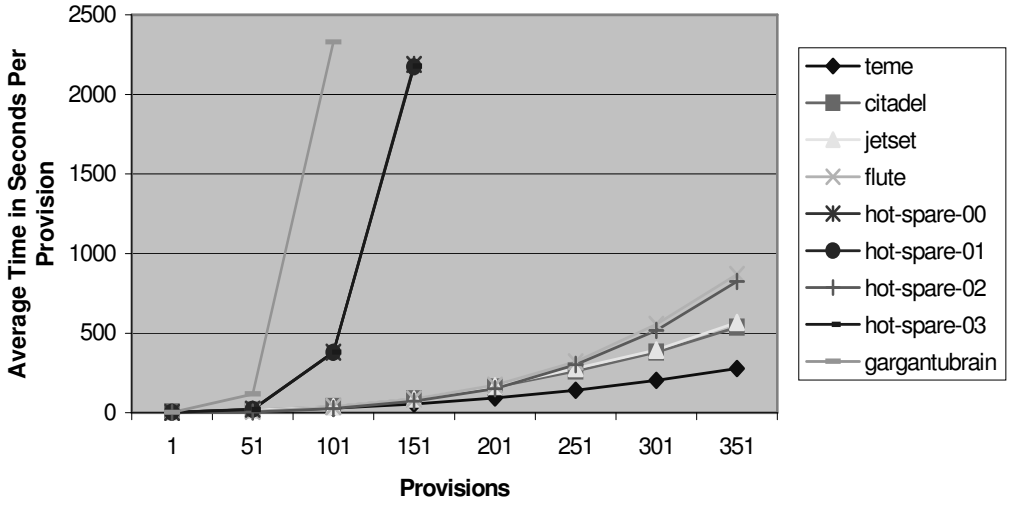


Figure 28: Average time, in seconds per provision, to insert and coverage-check provisions, comparing various installations

		Machine Name								
		teme	citadel	jetset	flute	hot-spare-00	hot-spare-01	hot-spare-02	hot-spare-03	gargantubrain
Number of Provisions	1	2.5	4.1	2.8	2.3	1.8	1.8	1.0	2.0	2.5
	51	12.0	15.6	15.1	9.7	21.0	20.7	4.9	20.9	117.2
	101	27.8	38.3	38.5	37.5	380.6	379.0	26.9	380.2	2329.9
	151	54.8	85.5	86.4	87.1	2189.3	2173.6	72.5	2177.1	
	201	92.3	162.2	167.9	173.4			151.3		
	251	142.2	260.6	278.4	321.3			303.9		
	301	203.8	380.9	392.9	553.9			517.7		
	351	279.3	536.8	567.0	867.7			825.5		
		Blank cells appear because tests on poorly-performing configurations were manually terminated at an early stage.								

Table 26: Average time, in seconds per provision, to insert and coverage-check provisions, comparing various installations

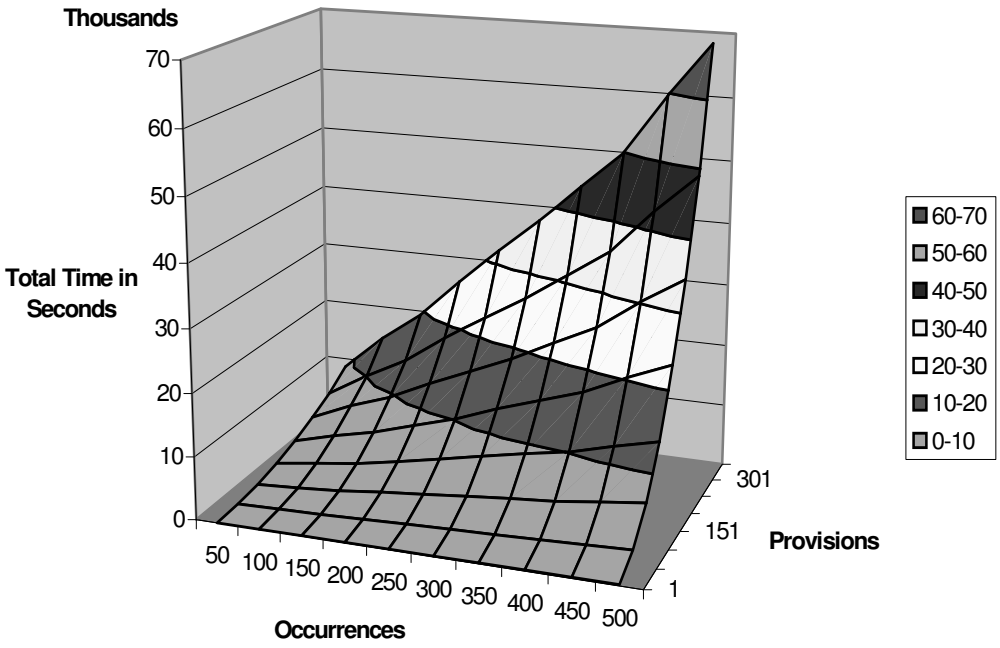


Figure 29: Total time, in seconds, to insert and coverage-check occurrences, on machine teme, for a varying number of stored provisions, and batch-size = 1

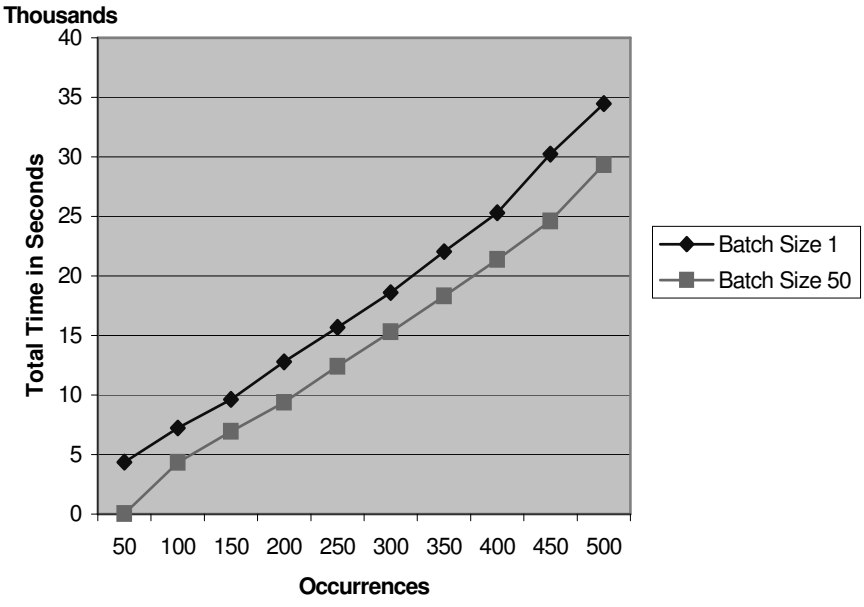


Figure 30: Total time, in seconds, to insert and coverage-check occurrences, on machine teme, for 251 stored provisions, and different batch-sizes

Number of Provisions	Number of Occurrences									
	50	100	150	200	250	300	350	400	450	500
1	20	37	54	74	94	117	140	164	193	219
51	232	389	587	863	1,092	1,339	1,579	1,793	2,113	2,412
101	659	1,108	1,626	2,411	3,051	3,736	4,405	4,990	5,884	6,715
151	1,522	2,527	3,407	4,748	6,082	7,536	8,866	10,118	11,955	13,591
201	2,771	4,574	6,086	8,152	10,155	12,694	14,941	17,164	20,370	23,181
251	4,343	7,207	9,618	12,788	15,686	18,609	22,032	25,313	30,233	34,459
301	6,124	10,104	13,624	18,227	22,478	26,472	31,031	35,621	42,797	48,932
351	8,577	14,463	19,382	25,860	31,956	37,627	43,884	49,989	59,647	68,285

Table 27: Total time, in seconds, to insert and coverage-check occurrences, on machine teme, for batch-size = 1

Shaded figures in Table 27 and Table 28 were used to plot Figure 30.

Number of Provisions	Number of Occurrences									
	50	100	150	200	250	300	350	400	450	500
1	5	20	31	45	61	78	95	111	131	154
51	10	244	401	598	865	1,100	1,344	1,564	1,777	2,097
101	12	664	1,119	1,671	2,486	3,197	3,932	4,582	5,206	6,076
151	14	1,527	2,497	3,382	4,689	6,023	7,474	8,714	9,986	11,712
201	16	2,789	4,556	6,116	8,104	10,149	12,682	14,794	17,028	20,061
251	19	4,324	6,935	9,379	12,410	15,300	18,301	21,387	24,615	29,331

Table 28: Total time, in seconds, to insert and coverage-check occurrences, on machine teme, for batch-size = 50

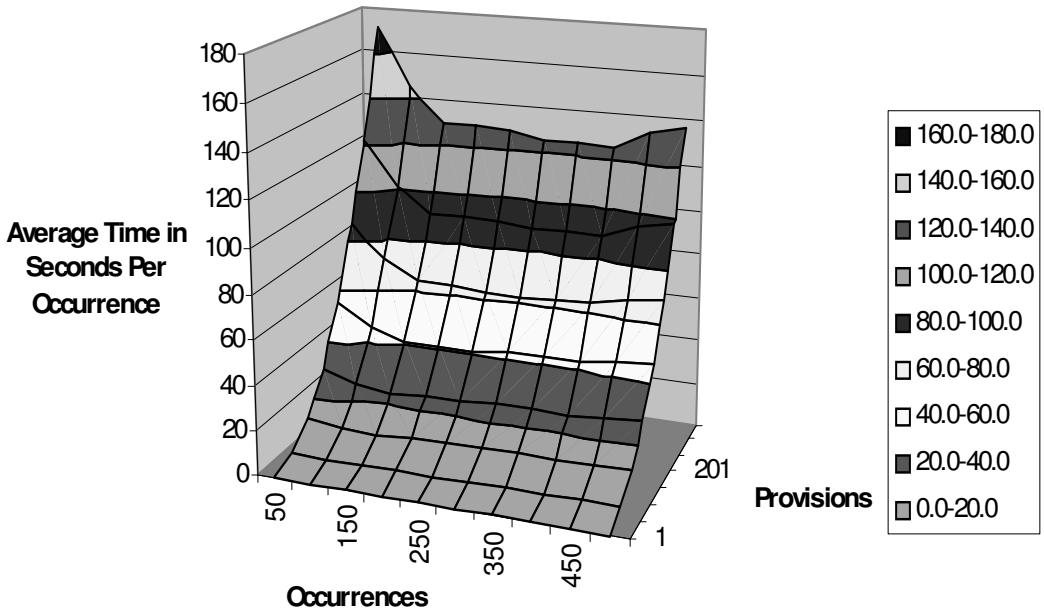


Figure 31: Average time, in seconds per occurrence, to insert and coverage-check occurrences, on machine teme, for a varying number of stored provisions, and batch-size = 1

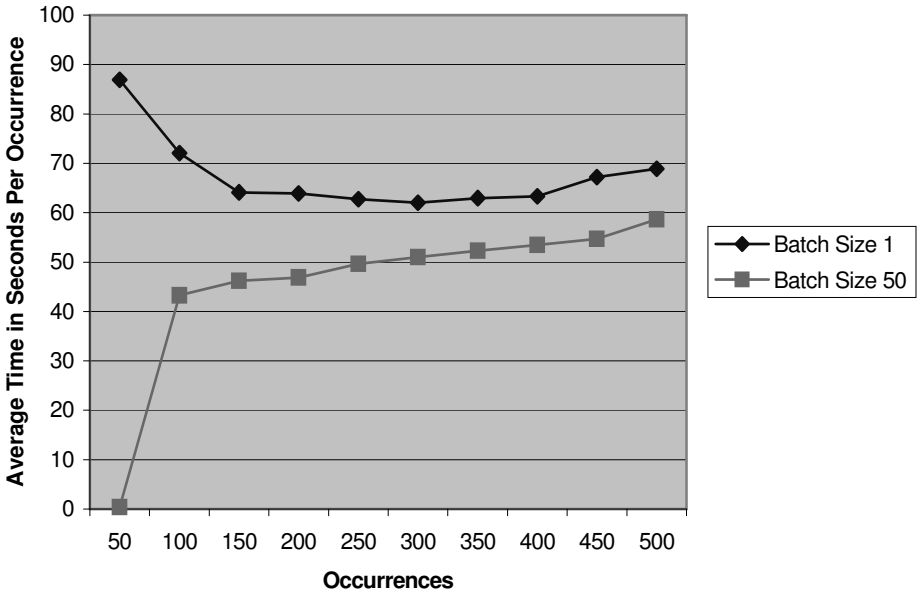


Figure 32: Average time, in seconds per occurrence, to insert and coverage-check occurrences, on machine teme, with 251 stored provisions, and varying batch-sizes

Number of Provisions	Number of Occurrences									
	50	100	150	200	250	300	350	400	450	500
1	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
51	4.6	3.9	3.9	4.3	4.4	4.5	4.5	4.5	4.7	4.8
101	13.2	11.1	10.8	12.1	12.2	12.5	12.6	12.5	13.1	13.4
151	30.4	25.3	22.7	23.7	24.3	25.1	25.3	25.3	26.6	27.2
201	55.4	45.7	40.6	40.8	40.6	42.3	42.7	42.9	45.3	46.4
251	86.9	72.1	64.1	63.9	62.7	62.0	62.9	63.3	67.2	68.9
301	122.5	101.0	90.8	91.1	89.9	88.2	88.7	89.1	95.1	97.9
351	171.5	144.6	129.2	129.3	127.8	125.4	125.4	125.0	132.5	136.6

Table 29: Average time, in seconds per occurrence, to insert and coverage-check occurrences, on machine teme, for batch-size = 1

Shaded figures in Table 29 and Table 30 were used to plot Figure 32.

Number of Provisions	Number of Occurrences									
	50	100	150	200	250	300	350	400	450	500
1	0.1	0.2	0.2	0.2	0.2	0.3	0.3	0.3	0.3	0.3
51	0.2	2.4	2.7	3.0	3.5	3.7	3.8	3.9	3.9	4.2
101	0.2	6.6	7.5	8.4	9.9	10.7	11.2	11.5	11.6	12.2
151	0.3	15.3	16.6	16.9	18.8	20.1	21.4	21.8	22.2	23.4
201	0.3	27.9	30.4	30.6	32.4	33.8	36.2	37.0	37.8	40.1
251	0.4	43.2	46.2	46.9	49.6	51.0	52.3	53.5	54.7	58.7

Table 30: Average time, in seconds per occurrence, to insert and coverage-check occurrences, on machine teme, for batch-size = 50

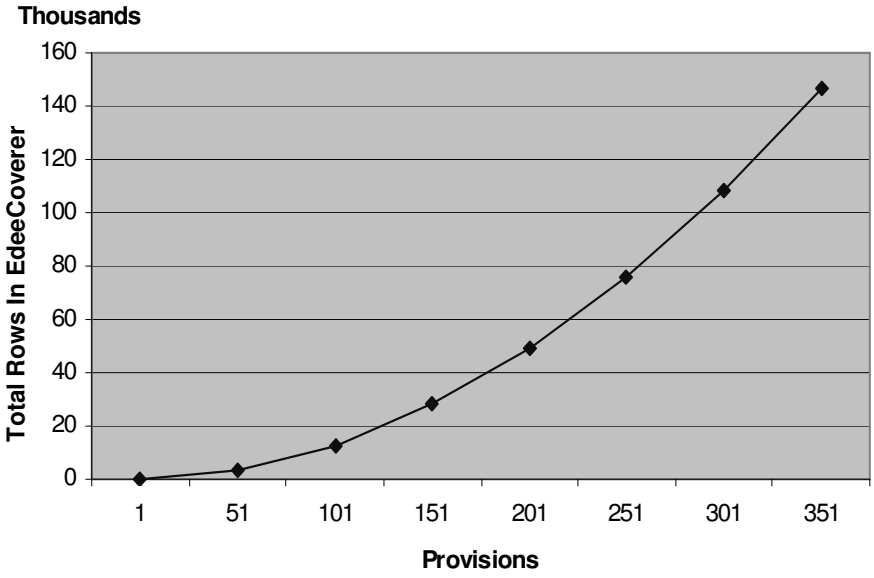


Figure 33: Total space for provisions, in rows of `EdeeCoverer` table (theoretical limit = n^2 where n = number of unique identifiers in database)

Number of Provisions	Total Space
1	7
51	2,972
101	12,572
151	28,031
201	49,007
251	75,563
301	107,949
351	146,659

Table 31: Total space for provisions, in rows of `EdeeCoverer` table

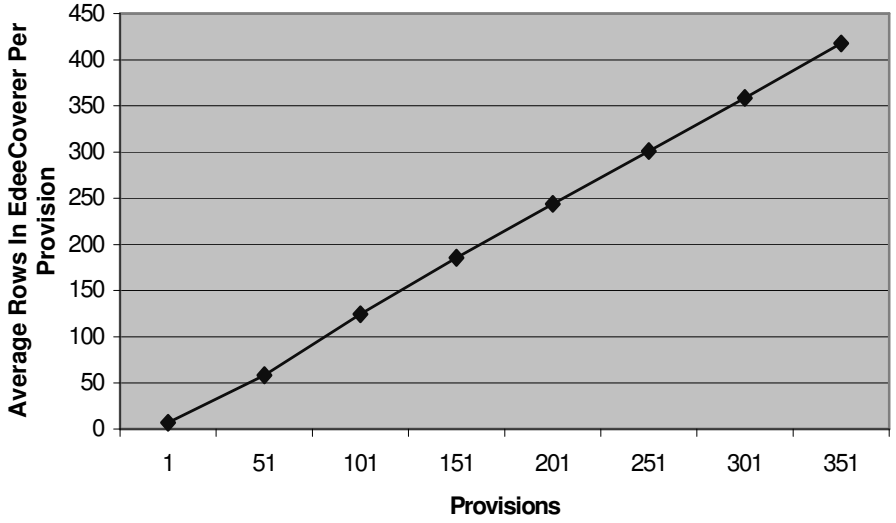


Figure 34: Average space for provisions, in rows of `EdeeCoverer` table (theoretical limit = *n* where *n* = number of unique identifiers in database)

Number of Provisions	Average Space Per Provision
1	7
51	58
101	124
151	186
201	244
251	301
301	359
351	418

Table 32: Average space for provisions, in rows of `EdeeCoverer` table

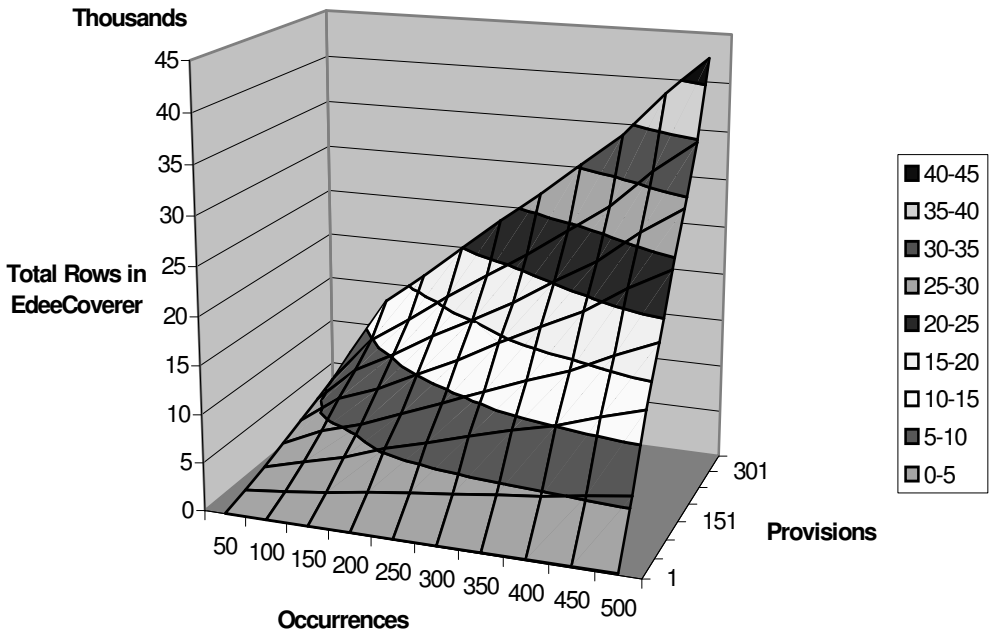


Figure 35: Total space for occurrences, in rows of `EdeeCoverer` table, for varying numbers of provisions (theoretical limit = n^2 where n = number of unique identifiers in database)

Number of Provisions	Number of Occurrences									
	50	100	150	200	250	300	350	400	450	500
1	26	33	45	64	75	90	101	120	147	167
51	687	1,183	1,717	2,423	2,972	3,544	4,115	4,595	5,308	5,921
101	1,433	2,620	3,699	5,272	6,500	7,766	8,990	10,053	11,514	12,816
151	2,331	4,326	5,578	7,325	9,036	10,832	12,559	14,109	16,217	18,043
201	3,276	6,276	7,911	10,050	12,075	14,488	16,721	18,811	21,637	24,010
251	4,150	7,913	10,200	12,781	15,051	17,390	20,169	22,708	26,174	29,030
301	5,125	9,873	12,699	15,753	18,570	21,279	24,316	27,326	31,387	34,727
351	6,502	12,969	16,325	19,925	23,451	26,780	30,351	33,838	38,556	42,486

Table 33: Total space for occurrences, in rows of `EdeeCoverer` table, for varying numbers of provisions

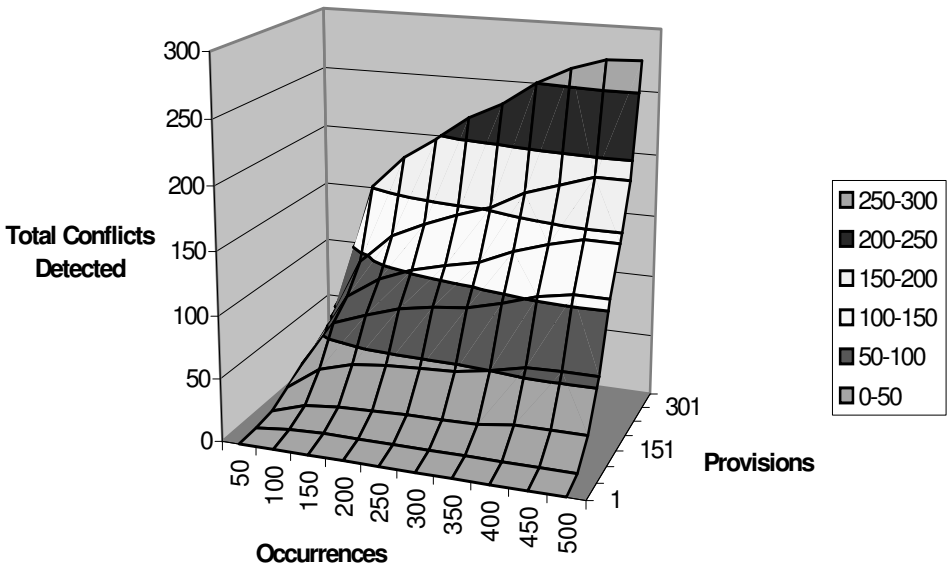


Figure 36: Total conflicts detected between individual prohibitions and obligations, for varying numbers of provisions and occurrences

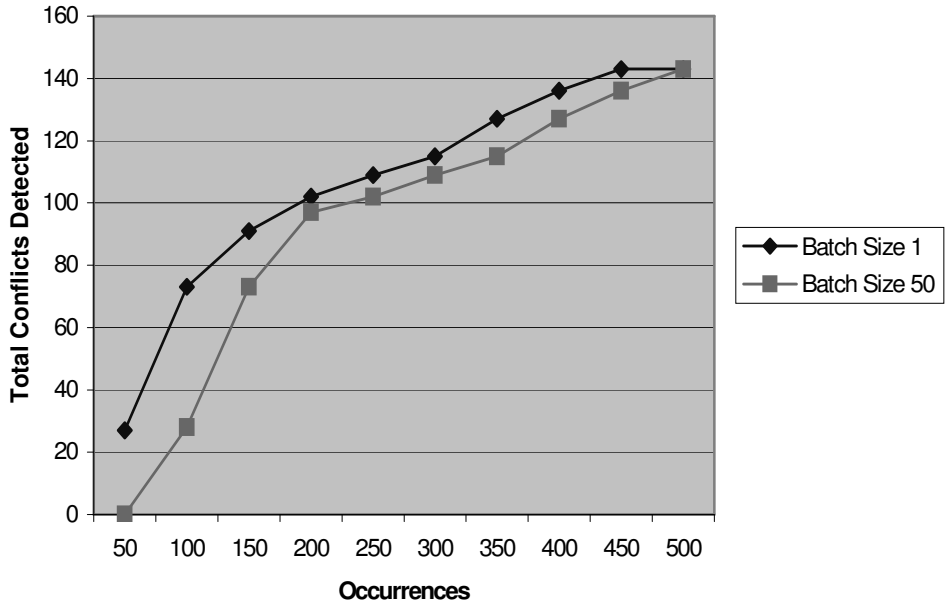


Figure 37: Total conflicts detected between individual prohibitions and obligations, for 251 stored provisions, comparing different batch-sizes

Number of Provisions	Number of Occurrences									
	50	100	150	200	250	300	350	400	450	500
1	0	0	0	0	0	0	0	0	0	0
51	1	4	5	5	5	5	5	5	5	5
101	3	12	15	16	17	17	18	22	22	22
151	12	31	39	42	44	45	50	56	57	57
201	21	60	70	79	84	87	94	103	108	108
251	27	73	91	102	109	115	127	136	143	143
301	37	94	119	132	143	152	167	176	185	185
351	57	151	179	197	217	231	250	264	273	274

Table 34: Total conflicts detected between individual prohibitions and obligations, for batch-size = 1

Shaded figures in Table 34 and Table 35 were used to plot Figure 37.

Number of Provisions	Number of Occurrences									
	50	100	150	200	250	300	350	400	450	500
1	0	0	0	0	0	0	0	0	0	0
51	0	1	4	4	4	4	4	4	4	4
101	0	3	12	16	16	17	17	18	22	22
151	0	11	27	36	38	40	41	46	52	53
201	0	22	60	75	78	83	86	93	102	107
251	0	28	73	97	102	109	115	127	136	143

Table 35: Total conflicts detected between individual prohibitions and obligations, for batch-size = 50

Analysis of Results

Our experiments illustrate that our algorithm supports small event histories, of a few hundred occurrences, in conjunction with a few hundred highly-conflicting legal provisions. Figure 36 above shows that EDEE successfully detects provisions coming into conflict with each other at run-time as new occurrences are added.

On the best performing machine, teme, it took an average of 280 seconds (almost 5 minutes) to individually insert and coverage-check a provision, when there were 351 provisions in the database. Similarly, it took an average of 136 seconds (just over 2 minutes) to insert and coverage check an occurrence, for 351 provisions and 500 stored occurrences. For less-conflicting sets of provisions, performance is likely to be substantially better than this.

As expected, Figure 30 (and, similarly, Figure 32) demonstrates that large batch sizes improve performance, but lead to a lag in the detection of conflicts (Figure 37), when compared to coverage-checking of each provision or occurrence individually.

Figure 27 indicates that our implementation scales better on Windows machines running Microsoft Access than on Linux machines running Postgres, and that tests executing on newer versions of Linux and Postgres yield better performance than those run on older releases of these products. It can be noticed from Figure 27 that no performance improvement was achieved through the use of multi-processor machines. Analysis of CPU usage, using Unix's `top` utility, revealed that this was because only one of the available processors would be employed (with typically 99% utilization), whereas the remaining processors experienced less than 0.5% utilization. Furthermore, on Unix, the database server (`postmaster`) accounted for more than 98% of CPU time, whilst the Java processes (with less than 2% usage) were mostly idle, waiting for results from the database.

While our results indicate that the current implementation is at present ill-suited to the real-time, high-throughput workflow applications for which it is intended, the approach is nevertheless clearly feasible for off-line, background performance and consistency assessment on medium-scale problems.

8.3.3 Comparative evaluation

Comparative evaluation of EDEE's occurrence monitor against similar systems is difficult, since EDEE's conceptual paradigm is markedly different from existing approaches.

EDEE underperforms against existing event monitoring and active database systems (§2.1) which monitor for low-level system and database events. The most efficient of these approaches may detect up to 600 events per second against 6 million subscriptions held in RAM [FJLP2001]. Such approaches treat events as transient, held in memory rather than stored in a database as in EDEE, with event attributes being simple string values. Event monitors are less expressive than expert systems and rule engine approaches (§2.2), which perform slowly as they indulge in more complex inferencing.

In the database view literature, the MiniCon containment-checking algorithm – demonstrably more scalable than the Bucket and Inverse-Rules approaches [PL2000] – is able to check a query against 1000 views (named queries) in less than one second. MiniCon is targeted at the data integration problem, and rewrites queries in terms of a maximal union of conjunctive queries over available views.

Expert system benchmarks use artificial intelligence problems with small, static rule sets. Typical commercial rule engines (page 26) use tests such as the NASA Monkeys & Bananas example as their benchmark. ILOG claims to solve this problem in less than 10 minutes [ILOG2001]. Problems mentioned in the academic expert systems literature include finding optimal seating arrangements for guests at a dinner party ('Manners' benchmark), labelling lines for simple scenes ('Waltz'), designing computer chip circuitry ('Weaver'), or determining the lowest-cost route plan for an aeroplane ('ARP') [BGLM91]. The basic 'Manners' benchmark uses only 8 rules, while the 'Waltz' benchmark uses 33 rules. The most complex of the benchmarks, 'Weaver', uses 1831 facts and 637 rules; it compiles to a rigid constraint network in 2-3 hours and executes in just over 11 minutes (on a SPARC Station 1+ with SunOS v4.0.3, using the OPS5c v1.09 compiler). Commercial contract evaluation scenarios, with large numbers of dynamic and conflicting rules cannot be

meaningfully assessed against these fixed object-pattern matching application benchmarks, which bear scant relation to workflow scenarios. For contract monitoring and checking, both object-pattern and dynamic pattern-pattern matching is required (§2.2.3, §8.1.4). In commercial trade environments, occurrences must be assessed against a large and periodically changing number of applicable provisions, and in the light of complex historical circumstances. While we are not yet close to the millisecond response times desired, we have demonstrated turnarounds of only a few minutes per provision or occurrence on complex contract assessment problems (§8.3.2).

Previous contract assessment work has been based on small, single contracts. Lee and co-workers [BLWW95, Lee88] have implemented a Petri-Net-based trade procedure executor, whilst Daskalopulu and colleagues [Das99, DDM2001] provide a conceptual framework for assessment of a small number of obligations (i.e. 2 or 3). Previous work has focused on developing small scale conceptual solutions to the problem, rather than on studying actual performance of implementations.

In the field of legal expert systems, Sergot et al. (op cit, page 172), using the relatively limited computing resources available at the time, showed in Prolog (see §2.2, §7.3) that the citizenship of an individual could be determined on the basis of approximately 500 rules of legislation. Performance figures were not provided.

We are not aware of any experimental studies of algorithms for ascertaining contract status or for determining the implications of business occurrences on contract consistency during workflow execution. We believe our experimental evaluation is itself a significant contribution that fills a void in previous work on this topic. We have shown for the first time that contract performance assessment and dynamic validation is viable on medium-scale problems with hundreds of provisions, small event histories, and a high proportion of run-time conflicts (§8.3.2).

8.4 Future Work

The previous section identified a variety of weaknesses in the current approach that highlight avenues for further exploration. In particular, performance improvements must be investigated (§8.4.1). Future work should also address usability concerns with the current approach – such as providing a simple user interface (§8.4.2) and automated completeness checks (§8.4.3) – as well as extending the approach to a distributed setting (§8.4.4 and §8.4.5).

8.4.1 Improving time and space efficiency

Our future work will look at improving the performance of the coverage-determination algorithm, with the eventual goal of assessing tens of thousands of occurrences against thousands of provisions within milliseconds.

Implementation of a native active database layer within the database kernel should yield significantly better performance (see §8.2.3), and will need to be examined.

Mechanisms for reducing the storage space requirement for covering relations (see §8.3.1), while preserving occurrence-semantics, should be investigated. A possibility is to tag transitively related data with sequence numbers as the data is added to the database; this would allow determination of the transitive closure to be achieved through a single query looking for data items tagged with greater, or lesser, values. This approach is suitable for totally-ordered data, but not for partially-ordered data. We will need to reconsider the triple store in search of a more efficient implementation with appropriate data structures.

8.4.2 Creating a user-friendly contract definition language

We have illustrated, at a storage level, the detailed underlying representation of occurrences, which are used to store workflow events (Chapter 3) and policies (Chapter 5). It should be remembered though that, for reasons of usability, the

interface to the system which stores the contracts and policies is more likely to be a constraining menu-driven interface or a simple English-like language which allows the user to work with the policies in a convenient and readable manner.

A contract definition language needs to be provided to allow business developers to input contract specifications. The mechanisms prototyped in EDEE can be exploited to store contracts and to determine when provisions apply and when a contract is fulfilled. However, these should be supplemented with simple contract presentations in order to express machine-parsable contracts in a form that humans can easily read and modify.

8.4.3 Assessing the completeness of contracts

An extension of EDEE would include algorithms and tools to ensure that contracts are fully specified. These tools would be applied automatically when a contract is defined or extended. Basic completeness checks include:

- *ensuring all role-players are specified*

  One mechanism for ensuring policy completeness is to detect unspecified role-players. For instance, in the specification "user privileges may change", the modal auxiliary "may" implies an occurrence of authorizing. As the authorized party is unspecified it appears that the policy is incomplete – the policy specifier has not specified who is authorized to change user privileges. To complete the specification, the policy specifier must therefore assign responsibility for this role. Similarly, occurrences implied by deverbative nouns in specifications must be bound to roles – the absence of necessary information associated with these occurrences may indicate incompleteness of the specification in that regard. For example, in "upon confirmation of registration, members will be able to log-in to a member's area of the site" the deverbative noun "confirmation" implies an occurrence of confirming whose actor should probably be specified for completeness (e.g. "confirmation by a customer services representative").

- *specification of consequences of violation*

  In cases where obligations and prohibitions are specified, the consequences of violation must generally be described. The analyst should be prompted to input such information. Such information should not, however, be mandatory as it is often the case that consequences of violation are to be decided only in the event of violation, as a party may not wish to minutely detail all penalties at the contracting stage.

8.4.4 Distributing contract and occurrence data to specialist nodes

The complexity of distributed organizations makes it difficult to enforce all organizational contracts centrally. Partitioning of related policies into stores monitored by specialist agents, and distribution of occurrences to interested and applicable specialist monitors, is a further optimization needed to exploit the principle of specialization and division of computational effort for contract management in medium and large enterprises. Specialist nodes should manage contracts for particular business units. Nodes must be aware of applicable specialist monitors so that they are able to forward relevant business occurrences to nodes interested in, and responsible for, assessing contract conformance in that sub-area of business.

8.4.5 Collating contracts from distributed nodes

The contracts governing the behaviour of an organisation and its agents constitute a dynamically changing set. Semi-autonomous subdivisions may independently define contracts with provisions that are mutually inconsistent, or that conflict with organisational or departmental goals. Independently specified policies must be collated from remote locations, analysed, and reconciled.

8.5 Summary

This chapter has summarized the strengths and weaknesses of our representation and implementation and identified opportunities for further research.

We gave a qualitative review of benefits (§8.1) and shortfalls (§8.2). In addition, we presented what we believe to be the first experimental evaluation of a contract performance assessment and dynamic validation algorithm (§8.3). It was established that, while performance is comparatively slow, our algorithm nonetheless is able to attend to complex, medium-scale problems within minutes.

As future work (§8.4), we will explore performance improvements. Implementation of the software in a distributed setting remains to be addressed, by providing facilities for distributing and collating provisions and occurrences; currently we store this information in a centralized database. We also plan to enhance usability by defining high-level business contract definition templates at user-interface level, and providing completeness checks.

This dissertation concludes with a summary of the contributions of the research.

Chapter 9

Conclusion: Contribution

This dissertation has proposed a method of approach for structuring the analysis and development of electronic commerce workflow applications. It has employed a new conceptual framework based on occurrences, and provided accompanying guidelines, information models, implementation structures, and algorithms. The work has highlighted the deficiencies of existing approaches to e-commerce application development (Chapter 2). It has addressed these issues by contributing:

9.1 A generic schema for storing and monitoring a history of business events and states (Chapter 3)

Traditional event monitors (§2.1) and policy-based systems (§2.3) monitor for recent, low-level events (§2.1.2, §2.1.3, §2.3.1). Events take constant values (typically `strings`, `integers`, or `dates`) as parameters.

We have defined a novel schema that affords uniformity of storage, detection, and querying for a *history of occurrences of business events and states* (§3.1). Our schema is able to represent *sentences with embedded propositions* including deontic sentences such as 'it is obliged / permitted / prohibited that …'. Such representation is achieved by

using *stored queries* (§3.2) that specify the criteria for the items and occurrences described in the embedded sentence. Our EDEE prototype permits parameter values (occurrence participants) to be described using queries, and provides *coverage-checking facilities* (§3.3) to analytically determine static and dynamic overlap between recorded queries.

9.2 A seamless application development approach catering for both analysis and implementation (Chapter 4)

Integration between the analysis and implementation phases of development has not been addressed by classical policy management systems (§2.3) and business contract architectures (§2.5), which focus almost entirely on technical facilities without regard for requirements analysis and formalization (§2.3.3).

The new techniques and tools we provide take account of both the analysis and implementation phases of the system development life cycle. To support the modelling process, we have described a set of analytic guidelines (§4.1-§4.6) that allow business analysts to systematically identify business workflow occurrences as well as normative occurrences such as the obligations, prohibitions, and powers imposed by a contract or user requirements specification. Development and deployment is achieved by storing the provisions identified from the analysis of user requirements, contracts, and regulations in an active database that is used to control workflow execution.

9.3 A representation schema for provisions of contracts, policies, and law (Chapter 5)

Current workflow systems (§2.4) take a task-dependency perspective (§2.4.1) and assume a rigid system of law for obligation creation (§2.4.2). Similarly, 'contracting' implementations (§2.5) do not explicitly represent or reason about the changing legal relations between parties. Policy-based approaches (§2.3) are limited to expressing action-based obligations; they omit treatment of several and collective obligations; they provide only synchronous invocation; and they hard-wire violation assessment conditions (§2.3.4). As with deontic logics (§2.6), obligation instances are neither individually identified nor distinguished from general obligation policies (§2.3.4, §2.6.4).

Our contribution has been to describe a valuable practical extension of Kimbrough's formal Disquotation Theory (§2.1.1), for the purpose of workflow automation. We made a number of revisions to Kimbrough's suggestions in the process: domain-specific roles replaced more ambiguous thematic roles (§3.1); provisions were relativized to an utterance (clause), rather than to the norm system as a whole (§5.1.1); occurrences of allocating supplanted the *Sake(...)* predicate which did not specify the allocation basis (§3.1, §5.6.1); violation states and obligation states were treated as separate entities (§5.6.2); permissions were extended to include the sense of vested liberty (§5.4); Hohfeld's notions of legal power, liability, disability, and immunity were added (§5.5, §5.3.2); a 'fulfilled' obligation was distinguished from a 'not violated' obligation (§5.6.1, §5.6.2); and, for obligations, we explicitly differentiated the party responsible to act from the party standing in surety, and the parties entitled to instigate or to receive recourse (§5.6.3).

Our approach looked at subjective assertions (§5.2), prohibitions (§5.3), permissions (§5.4), powers and liabilities (§5.5), and obligations (§5.6). One-shot and persistent rights were investigated (§5.5.1, §5.5.2). A consistent treatment of events and states for obligation fulfilment monitoring allowed the representation and

monitoring of both ought-to-do and ought-to-be obligations (§5.6.6). We accounted for conditional (§5.6.5) and secondary (§5.6.2), prima-facie and all-things-considered (§5.6.4, §5.6.9), and joint (§5.6.8) and several (§5.6.7) obligations.

9.4 A sophisticated mechanism for conflict detection and resolution (Chapter 6)

In some circumstances, more than one provision may be applicable, especially if contracts are modified. New provisions and data may introduce inconsistency. Rule-based approaches (§2.2) are often targeted at managing small, static rule-sets (§2.2.2). Conflict detection facilities are limited (§2.2.3) and only priority-based conflict resolution is provided (§2.2.4). Policy management systems (§2.3) attend to conflict avoidance rather than configurable resolution (§2.3.5). Deontic logics (§2.6) aim for consistent, conflict-free specifications (§2.6.2), and employ an idealistic notion of 'ought' (§2.6.1) that focuses on the standing obligations of the moral world, rather than the individual, dischargeable obligations found in business (§2.6.5).

We described a case-based policy instantiation mechanism that accounts for each individual norm instance's life cycle: creation and fulfilment, violation, or nullification (§5.6.9). *Contradictory provisions were dynamically detected* by analytically looking for overlap between occurrence descriptions (§3.3, §6.2). Our *conflict resolution* approach (§6.3) involved voiding identified obligation and prohibition instances, in accordance with specific clauses; it used defeasible, time-tagged conclusions (§6.4). The approach was able to make use of document structure information (such as position of a clause in a headed document or document-section) to locate *situated* clauses (§6.1) and select unambiguous clausal consequences in order to resolve conflicts.

9.5 An architecture for monitoring and enforcing a dynamically changing set of requirements (Chapter 7)

An important innovation was employing an incremental continuous query evaluation mechanism (§3.3) for the monitoring of occurrences against contractual provisions stored in tabular data schemas, using pervasive industrial technology (Java and relational databases). Reasoning about provision applicability and fulfilment or violation was done in the light of long-lived persistent occurrence histories. We specified a number of contract provision *monitoring* mechanisms – immediate (§7.1.1) and delayed (§7.1.2) detection – and *performance and enforcement* mechanisms – intervention (§7.2.1), prevention by refusal (§7.2.2), and prevention by construal (§7.2.3). The design proposed was verified through a prototype implementation, EDEE (§7.3). This implemented state-of-the-art support for legal-provision-based workflow automation that is not, to our knowledge, provided by any other application development infrastructure.

9.6 Summary of contribution

Embedding rules in opaque procedural code is untenable in enterprise scenarios where rules need to be externalized: that is, made observable, queryable, and modifiable. Requirements evolve rapidly and rules are volatile; the ability to update rules and determine which rules are applicable in cases where multiple rules pertain over time is crucial.

We have outlined an approach that caters for modelling, development, and deployment of business requirements in terms of contractual and regulatory provisions. A novel database representation and software wrapper were specified using a practical application scenario (§1.2): we demonstrated how to implement a

computational environment for electronic contract modelling, storage, analysis, and execution. Our approach was to store provisions in a database, and thereby exploit DBMS mechanisms for updating, locating, querying, and organizing provisions. This permits dynamic workflow configuration: the ability to adapt the workflow without stopping the application. Leveraging DBMS technology also gives the advantages of support for concurrent update, recovery, and replication. Business users are provided with facilities to determine, via database interrogation, which rules apply to a given component; this gives focused insights into the internal workings of complex applications. Effectively, the user requirements specification is stored in a database, can be modified and queried at run-time, and directly drives the workflow. Faster time-to-market and higher quality software can be expected, as specification and implementation share a common representation.

Contract-aware, occurrence-driven execution is an emerging paradigm for the automation of commercial organizational activities. It subsumes the notion of workflow, which has traditionally been the focus of computer science research in this general area. The ability to explicitly store, reason about, distribute and enforce contractual terms and provisions introduces synchronization between business requirements and application implementation. This realignment promises to advance the application development process by providing software primitives that mirror the fundamental conceptions of corporate law; the output of the formal analysis of business policies, contracts, and regulations then becomes usable for direct software implementation. The benefits of more correct and efficient business processes are far-reaching and potentially of great importance. Though much further work remains before we have an industrial-strength application, we believe our novel occurrence-based method, architecture, and software prototype implementation, have advanced the state-of-the-art in executable specification for e-commerce applications.

Appendix 1

Query Storage

This appendix gives details of the structure and syntax of stored queries (see §3.2). Table 36 below summarizes the query types, examples, and syntax. The various subsections of this appendix explore the major query types in more detail.

Name	Examples	Syntax in EDEEQL grammar[33]
Algebraic, Alphabetic, and Date Queries		

Greater than, Less than, Equal to	< 10 < 'Jeff' < 10 Nov 2001
	> 10 > 'Jeff' > 10 Nov 2001
	= 10 = 'Jeff' = 10 Nov 2001
		`< (less-than)`
		`> (greater-than)`
		`= (equal-to)`

Set-Theoretic Queries		
Concept Identification	= Steelmans	`= (identified-concept)`
Union	$Q_1 \cup \ldots \cup Q_n$	`(uniand) UNION (uniand) UNION … (uniand)`
Intersection	$Q_1 \cap \ldots \cap Q_n$	`(intersectand) INTERSECTION (intersectand) INTERSECTION … (intersectand)`
Difference	$Q_1 - Q_2$ or Q_1 but not Q_2	`(differor) - (differand) or (differor) BUT_NOT (differand)`
Empty Set	$\varnothing$	`EMPTY_SET`
Universal Set	U	`UNIVERSE`
Not / Complement	$(U - A_1)$	`UNIVERSE - (differand)`
Triplet Queries		
Participant Query	participants in role payer in paying1	`PARTICIPANTS IN ROLE (role) IN (occurrence)`
Occurrence Query	occurrences of paying where $25,000 is paid	`OCCURRENCES OF (type) WHERE (participant) IS/ARE (role)`
Role Query	roles of Steelmans in paying1	`ROLES (participant) IN (occurrence)`
Ordinal Queries		
Item in Position	1st of [occurrences of paying where $25,000 is paid] in ascending temporal order	`(position) OF (set) IN ASCENDING / DESCENDING (sequence) ORDER`
Inbuilt and User-Defined Operations		
Various	count(occurrences of paying where $25,000 is paid), max-possible-results(first of occurrences of paying), average(…), sum(…), min(…), max(…), …	`(operation-name) (set)`

Table 36: Types of query

[33] CriterionTypes are shown in round brackets, terminal symbols in `CAPITALS`. <, >, =, and - are also terminals. All queries are enclosed in square brackets '`[ … ]`'.

Note that, for queries that look up other stored queries, we use the syntax for Triplet Queries but we use the lexical item QUERYIDS in place of the lexical item OCCURRENCES, CRITERIONTYPE in place of ROLE, and VALUES in place of PARTICIPANTS. This is because, as shown in Figure 2 and Figure 3 (page 85), queries are stored in QueryID-CriterionType-Value tables rather than in the Occurrence-Role-Participant tables used to store occurrences.

A1.1 Algebraic Queries

Three basic types of algebraic query are catered for: equality, strictly less than, and strictly greater than. The three basic types of algebraic query are represented as shown in Table 37 below, which depicts the algebraic queries 'x = 6.2', 'x < 7', and 'x > 9' respectively (where x is the solution of the query). query901, query902, and query903 are identifiers for the three queries[34].

QueryID	CriterionType	Value	
query901	equal-to	6.2	(n1)[35]
query902	less-than	7	(n2)
query903	greater-than	9	(n3)

Table 37: Representation of basic algebraic queries: equality, strictly less than, and strictly greater than

Queries of the form 'greater than or equal', 'less than or equal', 'between' (with inclusive or exclusive upper and lower limits), and discontinuous ranges are represented using combinations of basic algebraic queries and set-theoretic queries described below. For example 'greater than or equal to' is represented as a union of a strictly greater than query and an equal to query, and a 'between' query with exclusive upper and lower limits is represented as the intersection of a strictly less than query involving the upper limit and a strictly greater than query involving the lower limit.

[34] As a convention in this Appendix, query numbers in the 900's are arbitrarily chosen, whereas other query numbers refer to queries mentioned in the main body of this document.

[35] As mentioned earlier, due to data-type restrictions, numbers are stored in the separate EdeeNumber table (see page 95). For readability we show the numbers directly; however, what appears in this column is really the number identifier: e.g. n1, n2, n3 where n1=6.2, n2=7, n3=9.

A1.2 Alphabetic Queries

Alphabetic queries are catered for in a similar manner to algebraic queries, though alphabetic queries of course cater for alphabetic comparison rather than numeric comparison. The three basic types of alphabetic query are represented as shown in Table 38 below, which depicts the queries 'x = Brian', 'x < Niki', and 'x > Peter' respectively (where x is the solution of the query).

QueryID	CriterionType	Value
query904	equal-to	'Brian' (s4)[36]
query905	less-than	'Niki' (s5)
query906	greater-than	'Peter' (s6)

Table 38: Representation of basic alphabetic queries: equality, strictly less than, and strictly greater than

As for algebraic queries (§A1.1), the basic types of alphabetic queries can be composed using set-theoretic queries.

A1.3 Set-Theoretic Queries

Set-theoretic queries for union, intersection, difference, identification, universal set, empty set, identification, negation, and cardinality (counting) are supported.

Union

Union queries return the items in *any* of the queries in the union. The criteria (arguments) to the union query can be referred to as 'uniands'. The representation of a general union query is as shown in Table 39 below:

[36] Again, due to data-type restrictions, symbols are stored in the separate `EdeeSymbol` table (see page 95). For readability we show the symbols directly; however, what appears in this column is really the symbol identifier: e.g. s4, s5, s6 where s4='Brian', s5='Niki', s6='Peter'.

QueryID	CriterionType	Value
query907	uniand	query601
	uniand	query602
	…	…

Table 39: Representation of a union query

Intersection

Intersection queries return the items in *all* of the queries in the intersection. The criteria (arguments) to the intersection query can be referred to as 'intersectands'. The representation of a general intersection query is as shown in Table 40 below:

QueryID	CriterionType	Value
query908	intersectand	query601
	intersectand	query602
	…	…

Table 40: Representation of an intersection query

Difference

Difference queries returns the items in first query *but not* in (i.e. *excluding* those items in) the second query set. The criteria (arguments) to the difference query can be referred to as the 'differor' and 'differand'. The representation of a general difference query is as shown in Table 41 below:

QueryID	CriterionType	Value
query909	differor	query601
	differand	query602

Table 41: Representation of a difference query

Universal Set A special query that returns all the concepts in the database. The universal set is represented simply by a specially assigned identifier.

Empty Set A special query that returns nothing. The empty set is represented simply by a specially assigned identifier.

Identification An identification query with the criterion `c1` returns the item identified as `c1` if that item exists in the database, and the empty set otherwise. Identification queries are used to allow identifiers to be wrapped inside queries so as to be nestable inside other queries. The representation of an identification query is as shown in Table 42 below:

QueryID	CriterionType	Value
query910	identified-concept	c1

Table 42: Representation of an identification query that returns `c1` if `c1` is in the database

Negation A negation ('not') query can be represented using the universal set as the differor and the negated set as the differand.

Cardinality A cardinality (counting) query is used to count the number of items fitting a description. A cardinality query takes a single criterion, which is a set of items to be counted; this set is generally specified using a query which describes the criteria for items falling into the set. This can be represented as shown in Table 43 below:

QueryID	CriterionType	Value
query911	counted	query601

Table 43: Representation of a query that counts the results of another query.

A1.4 Occurrence-Related Queries

Occurrence-related queries are queries that are solved by specifying any two of the occurrence, role, and participant columns and solving for the third. Occurrence-related queries look for identifiers in matching occurrence-role-participant triples; the values (or set or range of values) of two columns are specified in the criteria and the results are the contents of the third column in all rows where the two specified columns match. There are naturally three types of occurrence-related query:

Occurrence query An *occurrence query* specifies the occurrence type as well as the participant and its roles in that occurrence; all occurrence-identifiers of this type with this participant in that role are returned. Table 44 below shows an example.

QueryID	CriterionType	Value
query1	type	paying
query4	occurrence	query1
	participant	query2
	role	query3

Table 44: Representation of an occurrence query

Graphically, the 'search window' used to resolve an occurrence query looks like that shown in Table 45 below:

Occurrence	Role	Participant
	[match role criterion]	[match participant criterion]

Table 45: A graphical representation of the 'search window' used to resolve an occurrence query

Participant query

A *participant query* specifies the occurrence and role and can be represented as shown in Table 46 below; all participant-identifiers in this occurrence with this role are returned.

QueryID	CriterionType	Value
query7	occurrence	query5
	role	query6

Table 46: Representation of a participant query

Graphically, the 'search window' used to resolve a participant query looks like that shown in Table 47 below:

Occurrence	Role	Participant
[match occurrence criterion]	[match role criterion]	

Table 47: A graphical representation of the 'search window' used to resolve a participant query

Role query

A *role query* specifies the occurrence and participant and can be represented as shown in Table 48 below; all role-identifiers for the participant in the occurrence are returned.

QueryID	CriterionType	Value
query914	participant	query912
	occurrence	query913

Table 48: Representation of a role query

Graphically, the 'search window' used to resolve a role query looks like that shown in Table 49 below:

Occurrence	Role	Participant
[match occurrence criterion]		[match participant criterion]

Table 49: A graphical representation of the 'search window' used to resolve a role query

A1.5 Ordinal Queries

Ordinal queries are used to determine which item is in a certain position in a sequence, or to determine the position of an item in a certain sequence. In the former case (Table 50 below) the position number, set to search in, lookup direction, and sequence identifier (comparator function) must be specified; item identifiers are returned. In the latter case (not shown), the item identifier, set to search in, lookup direction, and sequence identifier must be specified; the position number is returned.

QueryID	CriterionType	Value
query915	position	1
	set	query914
	direction	ascending
	sequence	temporal

Table 50: Representation of an ordinal item-in-position query

A1.6 Nested Queries

Queries can be nested within each other. Nesting queries allows us to combine queries in order to specify complex criteria that are not otherwise expressible. Set-theoretic queries, occurrence-related, and conditional queries can nest queries of any other type within them. The values in the criteria used for set-theoretic, occurrence-related, and conditional queries are query identifiers.

A1.7 The Evaluation of Conditions

Complex conditions may be constructed using the operators `not`, `and`, `or`, `any-`*K* (i.e. cardinality / some specified number *K*), `all`, or `none`. These operators are not implemented in the usual truth-functional way of classical propositional logic since we are not dealing with propositions. Rather, they are implemented in a set-theoretic way since we are dealing with sets of occurrences, which are more fine-grained than propositions[37]. To evaluate conditions we select event occurrences and count them and then check if the count is correct (i.e. so-to-speak 'true' or false). This evaluation can be achieved using the various query types defined earlier; e.g. occurrence-related queries are used for selecting occurrences and cardinality queries are used for counting. Table 51 below demonstrates how a variety of complex conditions are evaluated in a set-theoretic, occurrence-centric manner.

[37] For example, instead of having merely the proposition *P* as in classical propositional logic, in an occurrence-centric view we have an identified occurrence (e.g. paying1) classified as being in the class *P*.

Appendix 1 - Query Storage

Example Condition	Set-Theoretic, Occurrence-Centric Evaluation (Implemented using the Occurrence-Centric Queries and Cardinality Queries defined earlier)
"you pay late"	Resolve "select 'late payings by you' occurrences". If the count of its results is greater than or equal to one, the condition is true, otherwise it is false.
"you do not[38] pay the correct person"	If the count of "select 'payments to the correct person' occurrences" yields zero (i.e. such an occurrence does not exist), the condition is true, otherwise it is false.
"you pay late and you are a regular customer"	Resolve "select 'you pay late' occurrences" and "select 'you are a regular customer' occurrences". Count the number of results of each of the selects. If none of the selects yield zero (i.e. zero of the selects yield zero results), the condition is true, otherwise it is false. Notice here that we are counting the count events (i.e. counting the count events that yield zero).
"you pay late or you pay too little"	If counting the results of the union of "select 'you pay late' occurrences" and "select 'you pay too little' occurrences" yields one or more, the condition is true, otherwise it is false.
"you do `any two or more of`[39]: pay late, change your travel date, or miss your flight"	At first glance "counting the results of (select the union of (select 'you pay late' occurrences and select 'you change your travel date' occurrences and select 'you miss your flight' occurrences)) yields 2 or more results" would seem the obvious way to evaluate this condition, but is not correct because it would not yield the correct result if you, for instance, changed your travel date twice but do neither of the other two things (i.e. did not pay late nor did you miss your flight). The correct implementation is: "counting the number of (selects which yield one or more results) yields 2 or more results", where the selects that are counted are "select 'you pay late' occurrences" and "select 'you change your travel date' occurrences" and "select 'you miss your flight' occurrences". Again, notice that you have to count your count events (i.e. count the counts that yield more than one) here; and then check that this count of counts yields two or more (as specified in the original condition "`any two or more of`..."). Notice that this condition would be difficult and very inefficient to implement in a traditional truth-theoretic manner, as propositional logic does not cater for counting and furthermore does not reify (individuate) occurrences so would not allow for the counting of occurrences; the condition would have to be implemented as a combination of "(you pay late and you change your travel date) or (you pay late and you miss your flight) or (you change your travel date and you miss your flight)" which is impractical for two reasons: - it would lead to a combinatorial explosion for large numbers of `k`, in '`any-k of`' expressions, and - it does not allow you to easily specify conditions like '`any X of`: …' where *X* is a variable computed at run-time.
you do `all/none of`: …	As for `any-k`, but replace *k* with the count of the number of different event-types listed (in the case of `all`) or with zero (in the case of `none`).

Table 51: Evaluating truth conditions in a set-theoretic, occurrence-centric manner

[38] Words shown in `Courier font` (e.g. `not`, `and`, `or`, `any-k`, etc.) are connectives for complex conditions.

[39] Notice that, because an `any-k` can be combined with an algebraic greater-than or less-than query, we can easily implement '`any k or more`' and '`any k or less`'.

Appendix 2

Coverage Checking Rules

This appendix describes in detail the rules for determining the relationships between queries. These rules themselves are effectively queries as they return results: the results are sets of queries which cover, are covered by, dirty, vacuum, are disjoint from, or partially overlap the specified query or described item. The following sub-sections define what is meant by the covering, dirtying, and vacuuming relationships between queries, and how to determine these relationships. Determination of disjointedness and partial-overlap relationships proceeds similarly.

A2.1 Relationships between queries

Queries may completely cover, dirty, vacuum, be disjoint from, or partially overlap other queries. These relationships are defined as follows:

- A query, A, **completely covers** (**is a superset of**) a query, B, if the appearance of any item *x* in the results of query B causes the appearance of that item *x* (which may be termed 'output dirt') in the results of the query A. The appearance of the item *x* in query B does not necessitate the partial re-evaluation of A since we can immediately conclude that a new result of A is *x*. Similarly, the disappearance of an item from the results of B causes its disappearance from A. The inverse of the **covering (superset)** relationship is the **covered (subset)** relationship. The 'completely covering' relationship is often referred to as query **containment** in the database literature. A query Q_1 is contained in the query Q_2, if the answer to Q_1 is a subset of the answer to Q_2 for *any* database instance [PL2000]. Query containment determination algorithms – such as Bucket, Inverse-Rules, and MiniCon – have previously been applied to the problems of view maintenance, query optimization, and data integration [PL2000]. As far as we are aware, ours is the first application of such techniques to contract assessment and analysis.
- A query is **dirtied** by an item if, when the item is added to the database, the item changes the *criteria* of the query, and therefore alters the results of the query. Dirtying, caused by the addition of application data, may bring two queries into or out of overlap at run-time. For example, upon the addition of the new clerks, John and Jeff, to the database, the query "salaries of **clerks**" is dirtied (and should be re-evaluated) as it must now also return "salaries of **John and Jeff**". A query, A, **dirties** a query, B, if the appearance of any item *x* (input dirt) in the results of query A alters the criteria of B and causes the appearance of an item *y* (where *y* may or may not be equal to *x*) or set of items *Y* (output dirt) in the results of the query B. To determine the new results of the dirtied query, substitute the input dirt for the dirtied criterion, to yield a partial re-evaluation

query. Then evaluate this partial re-evaluation query to obtain output dirt. See the applications of Rule A2.3.2 on page 88 for examples of the use of this technique.

- A query, A, **vacuums** a query, B, if the appearance of any item *x* in the results of query A may cause the *disappearance* of an item *y* (where *y* may or may not be equal to *x*) or set of items *Y* from the results of the query B. Again, a process of substitution and partial re-evaluation allows us to arrive at the changes in the results of B.
- A query, A, **is disjoint from** a query, B, if the appearance of any item *x* in the results of query A will certainly not cause the appearance of that item *x* in the results of the query B.
- A query, A, **partially overlaps** a query, B, if the appearance of an item *x* in a common subset of query A and query B, causes the appearance of that item *x* in the results of the query B, whilst the appearance of any item *x* in a subset of query A and query B which is not common to them both, will not cause the appearance of that item *x* in the results of the query B.

We say that an occurrence (or indeed any item) *fits a description* if it is *covered by a stored query*. Any new occurrence inserted in the occurrence store may be covered by a query, or may, via dirtying relationships, cause an existing occurrence or query to become covered by a query. For queries dirtied by another query, we need to do an **incremental evaluation (partial re-evaluation)** of the dirtied query to determine what new results and queries the dirtied query now covers. Since, for instance, queries may define a set of prohibited occurrences (§5.3), it is important to be able to determine the changes in query results as new occurrences are added, so that we may determine what occurrences become prohibited, and bring about violations, under a prohibition (§7.1.1).

A2.2 Determining which queries cover an item or query

Rule A2.2.1[40] An item or query (of any type) is covered by the universal set query (universe).

Rule A2.2.2 An item is covered by queries with matching type criteria.

Rule A2.2.3 Any query is covered by any query that is semantically identical to itself (i.e. contains the same criteria but different query identifiers). In order to factor out common sub-expressions and improve performance, a new query added to the database should be assigned the same query identifier as a semantically identical one already stored.

Rule A2.2.4 A query (of any type) is covered by any union query that has the query as one of its uniands.

Rule A2.2.5 Transitively, a query is covered by any query that covers any of the coverers of the query.

Rule A2.2.6 As covering relationships may also be stored, a query Q is covered by any query that is in the `Coverer` column in any row in the `EdeeCoverer` table (see pages 89, 95, and 216) in which Q is in the `Covered` column.

Rule A2.2.7 Any item or query is covered by any intersection query whose intersectands all cover the item or query; that is, the item or query is covered if no intersectands do not cover the item or query.

Rule A2.2.8 A number is covered by: numeric equal-to queries where the equal-to criterion equals the number; numeric less-than queries, where the less-than criterion is greater than number; and, numeric greater-than queries where the greater-than criterion is less than the number. Similarly for symbols and alphabetic queries.

[40] The query types and CriterionTypes mentioned in these rules are defined in Appendix 1.

Rule A2.2.9 A numeric equal-to query, Q, is covered by: numeric equal-to queries, where the equal-to criterion equals the equal-to criterion of Q; by numeric less-than queries where the less-than criterion is greater than the equal-to criterion of Q; and, by numeric greater-than queries where the greater-than criterion is less than the equal-to criterion of Q. A numeric less-than query, Q, is covered by numeric less-than queries where the less-than criterion is greater than the less-than criterion of Q. A numeric greater-than query, Q, is covered by numeric greater-than queries where the greater-than criterion is less than the greater-than criterion of Q. And similarly for alphabetic queries.

Rule A2.2.10 Any item (occurrence, role, or participant) is covered by concept-identification queries where the identified-concept criterion is identical to the occurrence-, role-, or participant-identifier. An identified concept is identical to another if their concept-identifiers are identical strings, or if a stored occurrence of being-identical relates the different identifiers for the concept. A concept-identification query is covered by another concept-identification query, if the identified-concept criteria of both queries are identical by either string-comparison or explicitly-stored being-identical relationships.

Rule A2.2.11 A concept-identification query is covered by any query that covers its identified-concept criterion.

Rule A2.2.12 A difference query is covered by its differor arguments. e.g. payments but not large payments is covered by payments.

Rule A2.2.13 An intersection query is covered by any of the queries in its intersectand set. e.g. full payments intersection large payments intersection late payments is covered by full payments, and by large payments, and by late payments.

Rule A2.2.14 An intersection query, Q, is also covered by any intersection query, P, if each of P's intersectands covers at least one of Q's intersectands. e.g. full payments intersection large payments intersection late payments is covered by full payments intersection large payments and by full payments intersection late payments and by large payments intersection late payments.

Rule A2.2.15 A union query is covered by any other queries that cover all of the uniands of the union query.

Rule A2.2.16 A participant query, Q, is covered by a participant query, P, if P's occurrence criterion covers Q's occurrence criterion, and P's role criterion covers Q's role criterion. Similarly for occurrence queries, and role queries.

Rule A2.2.17 An occurrence query is covered by its occurrence criterion.

Rule A2.2.18 An ordinal (sequence) query is covered by its set criterion. e.g. 1st [persons] in descending height order is covered by persons.

Rule A2.2.19 An item, y, in the range (i.e. output) of a programming language operation, Ω, is covered by $\Omega(\Omega^{-1}(y))$, where Ω^{-1} is the inverse operation of Ω. For example, consider the Java operation (in this case a user-defined abstract method) `symbolsStartingWith()` which maps a first letter to (the infinite set of) strings that start with that letter. This has the inverse operation `firstLetterOf()` which maps a string to its first letter[41]. Assume the item, y, we wish to coverage-check is the string '`Brian`'. We notice that, being a string, it is in the domain (input) of the inverse operation `firstLetterOf()` and, therefore, it is in the range (output) of the operation `symbolsStartingWith()`. Applying the inverse operation

[41] Notice that, because `symbolsStartingWith()` produces an infinite number of results per input in its domain, it cannot have a physical Java implementation, but it could be defined as an abstract method. In contrast, its inverse mapping method, `firstLetterOf()`, which produces only one result per input, can have a physical Java implementation. In general, computing the inverse operation for a given operation is non-trivial, and we provide no facilities for automated generation of operation inverses. It is assumed that the developer manually specifies the name and definition of the inverse operation when defining an operation, and that the system provides a commonly used set of operations and their inverses. Automatic determination of inverse methods is a potential area of future research.

`firstLetterOf()` to '`Brian`' we get the result '`B`'. We therefore know that the operation `symbolsStartingWith()` when invoked using the specific parameter '`B`', covers (among other things) the string '`Brian`'. That is, we have used the inverse operation, `firstLetterOf()`, to show that '`Brian`' is *covered by the description* `symbolsStartingWith(`B′)`. This method of using inverse operations to find descriptions (operations and the specific parameter values) that cover an item can be generally applied. Furthermore, as far as we are aware, this is a novel mechanism of finding descriptions for items.

Notice that the coverage-determination process is the reverse of the traditional query-resolution process used by query executors. Query execution can be implemented through the is-covered-by (i.e. is-a-subset-of) relationship, which is the inverse relationship of covers (i.e. is-a-superset-of).

A2.3 Determining which queries or items are dirtied by a query

Rule A2.3.1 A query, Q, dirties any difference queries that have the query, Q, as its differor.

Rule A2.3.2 A query, Q, dirties any triplet query that has the query, Q, as its occurrence criterion, role criterion, or participant criterion.

Rule A2.3.3 A query, Q, dirties any intersection query that has the query, Q, as its intersectand criterion.

Rule A2.3.4 An ordinal query or an inbuilt or user-defined operation is dirtied by any changes to the set criterion defining its input set. Incremental re-evaluation mechanisms vary depending on the operation: a count may be incrementally re-evaluated by adding the total number of new items to the last count, thereby saving a full recount. Similarly for a sum, where the sum of new values is added to the last total. An average can be incrementally re-evaluated by the formula: ((last_average x last_count)+total_of_new_values)÷(last_count+number_of_new_values).

A2.4 Determining which queries or items are vacuumed by a query

Difference queries are **vacuumed** by their differands – that is, when a differand becomes dirty, the parent difference query is immediately *vacuumed* as it no longer contains the results that are newly produced in its differand argument. So, when the differand of a difference query becomes dirty, the dirt is removed (i.e. concepts are sucked out from) the results of the parent difference query, which is then said to be **vacuumed**. This leaves a vacuum because some of the results that used to be in the result set of the vacuumed query are no longer there. This vacuum propagates to queries which *cover* the vacuumed query – i.e. those queries no longer cover items in the vacuum.

Bibliography

Self-references (**[AB2000]**, **[AB2001a]**, **[AB2001b]**, **[AB2001c]**, **[AB2001d]**, **[AB2002a]**, **[AB2002b]**, **[AEB2002a]**, **[AEB2002b]**, **[AEB2002c]**, **[AK2002]**, **[DA99]**) are listed in the **Publications** section on page xi.

[AAD97] Allwood J, Andersson L-G, and Dahl O. *Logic in Linguistics*. Cambridge University Press. Cambridge, UK. 1997.

[AGMW97] Adelberg B, Garcia-Molina H, and Widom J. The STRIP Rule System for Efficiently Maintaining Derived Data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. pp. 147-158. Tucson, AZ. May 1997.

[AK96] Ayres R and King PJH. Querying Graph Databases Using a Functional Language Extended with Second Order Facilities. In Morrison R and Kennedy J (eds): *Advances in Databases Systems, Proceedings of 14th British National Conference on Databases (BNCOD14)*. pp. 189-203. Edinburgh, United Kingdom. July 3-5, 1996.

[All95] Allen J. *Natural Language Understanding*. 2nd Edition. Benjamin Cummings. Menlo Park, CA. 1995.

[And58] Anderson AR. A Reduction of Deontic Logic to Alethic Modal Logic. *Mind* 67. pp. 100-103. 1958.

[And62] Anderson AR. Logic, Norms, and Roles. *Ratio*. 4(36). pp. 36-49. 1962.

[ATG2001] ATG Dynamo. http://www.atg.com/. 2001.

[Aus76] Austin JL. *How to do things with words*. 2nd Edition. Oxford University Press. 1976.

[Bac86] Bach E. The Algebra of Events. *Linguistics and Philosophy*. 9. pp. 5-16. 1986.

[BBH97] Bacon J, Bates J, and Halls D. Location oriented multimedia. *IEEE Personal Communications*. 4(5). pp. 48-57. October 1997.

[BBHM95] Bacon J, Bates J, Hayton R, and Moody K. Using events to build distributed applications. *Proceedings of the 2nd International Workshop on Services in Distributed and Network Environments*. Whistler, BC. pp. 148-155. 1995.

[BDLT2000] Bons RWH, Dignum F, Lee RM, and Tan Y-H. A Formal Analysis of Auditing Principles for Electronic Trade Procedures. *International Journal of Electronic Commerce*. 5(1). Fall 2000.

[BEA2001] BEA WebLogic Server. http://www.bea.com/products/weblogic/server/index.shtml. 2001.

[Ben88] Bennett J. *Events and their Names*. Oxford University Press. Oxford, UK. 1988.

[BFA99] Bertino E, Ferrari E, and Atluri V. The Specification and Enforcement of Authorization Constraints in Workflow Management Systems. *ACM Transactions on Information and Systems Security*. 2(1). pp. 65-104. February 1999.

[BFL96] Blaze M, Feigenbaum J, and Lacy J. Decentralized Trust Management. *Proceedings of the IEEE Conference on Security and Privacy*. Oakland, CA. 1996.

[BGLM91] Brant DA, Grose T, Lofaso B, and Miranker DP. Effects of Database Size on Rule System Performance: Five Case Studies. *Proceedings of the 17th International Conference on Very Large Data Bases*. 1991.
See also: http://www.cs.utexas.edu/ftp/pub/ops5-benchmark-suite/HOW.TO.USE.

[BHB96] Bates J, Halls D, and Bacon J. A Framework to Support Mobile Users of Multimedia Applications. *ACM Mobile Networks and Nomadic Applications (NOMAD)*. 1(4). pp. 409-419. 1996.

[BHMM2001] Bacon J, Hombrecher A, Ma C, Moody K, and Yao W. Event Storage and Federation using ODMG. *9th International Workshop on Persistent Object Systems, Design, Implementation and Use (POS9)*. Lillehammer, Norway. September 2000. Lecture Notes in Computer Science 2135. pp. 265 – 281. Springer-Verlag. Berlin, Germany. 2001.

[BHMM2002] Bacon J, Hombrecher A, Ma C, Moody K, and Pietzuch P. Building Event Services on Standard Middleware. *Software Practice and Experience*. John Wiley & Sons. To appear.

[BJPW99] Beugnard A, Jézéquel J-M, Plouzeau N, and Watkins D. Making Components Contract Aware. *IEEE Computer*. pp. 38-45. July 1999.

[Bla2000] Blaze Software. Blaze Advisor Technical White Paper.
Available from: http://www.blazesoft.com/products/docrequest.html
Accessed on: 1 March 2000.

[Bla2001] Blaze M. *Using the KeyNote Trust Management System*. March 2001. http://www.crypto.com/trustmgt/kn.html.

[BLM2001] Bacon J, Lloyd M, and Moody K. Translating role-based access control policy within context. In [SLL2001]. pp. 107–119.

[BLWW95] Bons RWH, Lee RM, Wagenaar RW, and Wrigley CD. Modelling Inter-organizational Trade Procedures Using Documentary Petri Nets. *Proceedings of the Hawaii International Conference on System Sciences*. 1995.

[BM93] Brant DA and Miranker DP. Index Support for Rule Activation. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. pp. 42 – 48. 1993.

[BMBH2000] Bacon J, Moody K, Bates J, Hayton R, Ma C, McNeil A, Seidel O, and Spiteri M. Generic support for distributed applications. *IEEE Computer*. 33(3). pp. 68–77. March 2000.

[Bro96] Brown MA. Doing As We Ought: Towards a Logic of Simply Dischargeable Obligations. In Brown MA and Carmo J (eds): *Deontic Logic, Agency, and Normative Systems: Proceedings of the Third International Workshop on Deontic Logic In Computer Science (DEON'96)*. Sesimbra, Portugal. Springer. 1996.

[Bro2000] Brown MA. Conditional Obligation and Positive Permission for Agents in Time. *Nordic Journal of Philosophy*. 5(2). pp. 83-112. 2000.

[Bro2001] Brokat Technologies AG. *White Papers about Brokat Advisor*. August 2001. http://www.brokat.com/cgi-bin/wpnav.cgi

[BRS2001] Business Rules Solutions, LLC. *BRS RuleSpeak* and *BRS RuleTrack*. August 2001. Available at: http://www.brsolutions.com/{rulespeak_download, ruletrack}.shtml

[BSHB98] Bates J, Spiteri MD, Halls D, and Bacon J. Integrating Real-World and Computer-Supported Collaboration in the Presence of Mobility. *Proceedings of IEEE Workshops on Emerging Technologies in Collaborative Environments (WET'ICE 1998) Workshop on Collaboration in the Presence of Mobility*. Stanford, CA. June 1998.

[BvdR95] Burg JFM and van de Riet RP. COLOR-X: Object Modelling Profits from Linguistics. *In Proceedings of the 2nd International Conference on Building and Sharing of Very Large-Scale Knowledge Bases (KB&KS'95)*. Enschede, The Netherlands. 1995.

[Cam2001] Cameron GD. The Legal Infrastructure for E-commerce: What do we have and what do we need? *International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR2001)*. L'Auqila, Italy. ISBN:88-85280-61-7. August 2001.

[CC95] Cholvy L and Cuppens F. Solving Normative Conflicts by Merging Roles. *Proceedings of the Fifth International Conference on Artificial Intelligence and the Law (ICAIL'95)*. Washington, DC. pp. 201-209. May 1995.

[CC98] Cholvy L and Cuppens F. Reasoning about norms provided by conflicting regulations'. In Prakken H and McNamara P (eds): *Norms, Logic, and Information Systems: New Studies in Deontic Logic and Computer Science*. Amsterdam. IOS Press. pp. 247-264. 1998.

[CCS94] Collet C, Coupaye T, and Svensen T. NAOS: Efficient and modular reactive capabilities in an Object-Oriented Database System. *20th International Conference on Very Large Databases*. Santiago, Chile. September 1994.

[CCS97] Cholvy L, Cuppens F, and Saurel C. Towards a logical formalization of responsibility. *Proceedings of the Sixth International Conference on Artificial Intelligence and the Law*. Melbourne, Australia. ACM Press. pp. 233-242. 1997.

[CDDF98] Ceri S, Di Nitto E, Discenza A, Fuggetta A, and Valetto G. *DERPA: A Generic Distributed Event-based Reactive Processing Architecture.* Technical Report. CEFRIEL – Politecnico di Milano. March 1998.

[CDMR2001] Cole J, Derrick J, Milosevic Z, and Raymond K. Author Obliged to Submit Paper before 4th July: Policies in an Enterprise Specification. In [SLL2001]. pp. 1-17.

[CDNF2001] Cugola G, Di Nitto E, and Fuggetta A. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering.* 27(9). pp. 827-850. September 2001.

[CDTW2000] Chen J, DeWitt D, Tian F, and Wang Y. NiagarCQ: A Scalable Continuous Query System for Internet Databases. *Proceedings of the 2000 ACM SIGMOD International Conference on the Management of Data.* Dallas, TX. pp. 379-390. May 2000.

[Che80] Chellas BF. Deontic Logic and Conditional Logic. Chapters in Chellas BF. *Modal Logic: An Introduction.* Cambridge University Press. pp. 190-203, 268-277. 1980.

[CG89] Carriero N and Gelernter D. Linda in context. *Communications of the ACM.* 32(4). pp. 444-458. April 1989.

[CIDE95] *Cambridge International Dictionary of English.* Cambridge University Press. 1995.

[CKAK93] Chakravarthy S, Krishnaprasad V, Anwar E, and Kim S-K. Anatomy of a Composite Event Detector. *Technical Report UF-CIS-TR-93-039.* Department of Computer and Information Sciences, University of Florida. Gainesville, FL. December 1993.

[CL2001] Chomicki J and Lobo J. Monitors for History-Based Policies. In [SLL2001]. pp. 107–119.

[CLN2000] Chomicki J, Lobo J, and Naqvi S. A Logic Programming Approach to Conflict Resolution in Policy Management. *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR'2000).* Breckenridge, CO. Morgan Kaufman. pp. 121-132. April 2000.

[CLZ2000] Cabri G, Leonardi L, Zambonelli F. Mobile-Agent Coordination Models for Internet Applications. *IEEE Computer*. pp. 82-89. February 2000.

[CM94a] Chakravarthy S and Mishra D. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*. 14(1). November 1994.

[CM94b] Clocksin WF and Mellish CS. *Programming in Prolog*. 4th Edition. Springer-Verlag. Berlin, Germany. 1994.

[CP2000] Carmo J and Pacheco O. Logics for Modelling Businesses and Agents Interaction. *International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR2000)*. L'Auqila, Italy, ISBN 88-85280-52-8. July-August 2000.

[Cru2000] Cruse A. *Meaning in Language: An Introduction to Semantics and Pragmatics*. Oxford University Press. Oxford, UK. Chapter 16, pp. 331-346. 2000.

[CRW98] Carzaniga A, Rosenblum DS, and Wolf AL. Design of a Scalable Event Notification Service: Interface and Architecture. *Technical Report CU-CS-863-98*. Department of Computer Science, University of Colorado. Boulder, CO. August 1998.

[CSvdR99] Caminada MWA, Steuten AAG, and van de Riet RP. An Evaluation of Linguistically Based Modelling Approaches on the Basis of the EANCOM EDI Standard. *The Language Action Perspective*. 1999.

[CT96a] Checkland PB and Tsouvalis C. *Reflecting on the SSM: The Dividing Line Between 'Real World' And 'Systems Thinking World'*. Working Paper Number 3. The Centre for Systems and Information Sciences. University of Humberside. 1996.

[CT96b] Checkland PB and Tsouvalis C. *Reflecting on the SSM: The Link Between Root Definitions and Conceptual Models*. Working Paper Number 5. The Centre for Systems and Information Sciences. University of Humberside. 1996.

[CTVR98] Ciancarini P, Tolksdorf R, Vitali F, Rossi D, and Knoche A. Coordinating Multi-agent Applications on the WWW: A Reference Architecture. *IEEE Transactions on Software Engineering*. 24(5). pp. 362-375. May 1998.

[Dam2002] Damianou NC. *A Policy Framework for Management of Distributed Systems*. PhD Thesis. Department of Computing, Imperial College, University of London. 2002.

[dAMW96] d'Altan P, Meyer JJCh, and Wieringa RJ. An integrated framework for ought-to-be and ought-to-do constraints. *Artificial Intelligence and Law*. 4. pp. 77-111. 1996. Follow-on version as at 22 January 1998.

[Das97] Daskalopulu A. Logic-based Tools for Legal Contract Drafting: Prospects and Problems. *Proceedings of the 1st Logic Symposium*. University of Cyprus Press. pp. 213-222. 1997.

[Das98] Daskalopulu A. Legal Contract Drafting at the Micro Level. *Law in the Information Society: 5th International Conference of the Institute of Legal Documentation (IDG) of the Italian National Research Council*. Florence, Italy. December 1998.

[Das99] Daskalopulu A. *Logic-Based Tools for the Analysis and Representation of Legal Contracts*. PhD Thesis. Department of Computing, Imperial College, University of London. 1999.

[Das2000] Daskalopulu A. Modelling Legal Contracts as Processes. *DEXA Workshop on Legal Information Systems Applications (LISA200)*. Greenwhich. September 2000.

[Dat2000] Date CJ. *What Not How: The Business Rules Approach to Application Development*. Addison-Wesley. Reading, MA. 2000.

[Dav80] Davidson D. *Essays on Actions and Events*. Clarendon Press, Oxford. 1980.

[Dav96] Davis T. *Lexical Semantics and Linking in the Hierarchical Lexicon*. PhD Thesis. Stanford University, Department of Linguistics. pp. 17-69. 1996.

[DB2001] Dimitrakos T and Bicarregui J. Towards a Framework for Managing Trust in e-Services. *Proceedings of the Fourth International Conference on Electronic Commerce Research*. Volume 2. Dallas, TX. ATSMA, IFIP, INFORMS. pp. 360-381. 2001.

[DDKL2001] Dan A, Dias DM, Kearney R, Lau TC, Nguyen TN, Parr FN, Sachs MW, and Shaick HH. Business-to-business integration with tpaML and a business-to-business protocol framework. *IBM Systems Journal*. 40(1). 2001.

[DDLS2001] Damianou N, Dulay N, Lupu E, and Sloman N. The Ponder Policy Specification Language. In [SLL2001]. pp. 18-38.

[DDM2001] Daskalopulu A, Dimitrakos T, and Maibaum TSE. E-Contract Fulfilment and Agents' Attitudes. *Proceedings ERCIM WG E-Commerce Workshop on the Role of Trust in E-Business.* Zurich. October 2001.

[Del2000] Dellarocas C. Contractual Agent Societies: Negotiated shared context and social control in open multi-agent systems. *Workshop on Norms and Institutions in Multi-Agent Systems at the 4th International Conference on Multi-Agent Systems (Agents-2000).* Barcelona, Spain. June 2000.

[DFGG2000] Dittrich KR, Fritschi H, Gatziu S, Geppert A, and Vaduva A. SAMOS in Hindsight: Experiences in Building an Active Object-Oriented DBMS. *Technical Report 2000.05.* Database Technology Research Group. Department of Information Technology, University of Zurich. 2000.

[DK97] Dignum F and Kuiper R. Combining Dynamic Deontic Logic and Temporal Logic for the Specification of Deadlines. *In Proceedings of the 30th Hawaii International Conference on Systems Sciences (HICSS'97).* Hawaii. 1997.

[DK98] Dignum F and Kuiper R. Specifying Deadlines with Dense Time Using Deontic and Temporal Logic. *International Journal of Electronic Commerce.* 3(2). pp. 67-86. Winter 1998-99.

[DK99] Dellarocas C and Klein M. Designing robust, open electronic marketplaces of Contract Net agents. *Proceedings of the 20th International Conference on Information Systems (ICIS).* Charlotte, NC. December 1999.

[DLSD2001] Dulay N, Lupu E, Sloman N, and Damianou N. A Policy Deployment Model for the Ponder Language. *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management (IM'2001).* Seattle, WA. May 2001.

[DM2001] Daskalopulu A and Maibaum TSE. Towards Electronic Contract Performance. *Legal Information Systems and Applications (LISA), 12th Conference and Workshop on Database and Expert Systems Applications.* IEEE CS Press. pp. 771-777. 2001.

[Dow87] Dowson M. ISTAR and the Contractual Approach. *Proceedings of the 9th International Conference on Software Engineering (ICSE 1987).* Association for Computing Machinery. Monterey, CA. pp. 287-288. 1987.

[DS95] Daskalopulu A and Sergot M. A Constraint-Driven System for Contract Assembly. *Proceedings of the 5th ACM International Conference on AI and Law (ICAIL-95).* ACM Press. pp. 62-70. 1995.

[DS97] Daskalopulu A and Sergot MJ. The Representation of Legal Contracts. *AI and Society.* 11 (1/2). pp. 6-17. 1997.

[DS2002] Daskalopulu A and Sergot MJ. Computational Aspects of the FLBC Framework. *Decision Support Systems.* 33(3). Special Issue on Formal Modelling in E-Commerce. pp. 267-290. July 2002.

[DSvdR2000] Dehne F, Steuten AAF, and van de Riet RP. WordNet++: A Lexicon Supporting the Color-X Method. *In Proceedings of the 5th International Conference on Application of Natural Language to Information Systems (NLDB'2000).* Versailles, France. 2000.

[DW95] Dignum F and Weigand H. Communication and deontic logic. In Wieringa R and Feenstra R (eds): *Information Systems, Correctness and Reusability.* World Scientific. Singapore. pp. 242-260. 1995.

[DWV96] Dignum F, Weigand H, and Verharen E. Meeting the deadline: on the formal specification of temporal deontic constraints. *Proceedings of the International Symposium on Methodologies for Intelligent Systems.* pp. 243-252. 1996.

[ELW94] Embley DW, Liddle SW, and Woodfield SN. Attributes: Should We Eliminate Them from Semantic and Object-Oriented Data Models? *Proceedings of the Computer Science Conference.* ACM Press. New York, NY. pp. 340-347. 1994.

[ESP99] Eiter T, Subrahmanian VS, and Pick G. Heterogeneous Active Agents, I: Semantics. *Artificial Intelligence.* 108(1-2). pp. 179-255. 1999.

[FF94] Fromherz MPJ and Fuchs N. *Software Development Based on Executable Specifications and Transformations.* Report 94.07. Institut für Informatik, Universität Zürich. 1994.

[FH2001] Friedman-Hill EJ. *Jess, the Expert System Shell for the Java Platform*. SAND98-8206. Sandia National Laboratories. Livermore, CA. August 2001. Available at: http://herzberg.ca.sandia.gov/jess/

[FJ91] Flood RL and Jackson MC. *Creative Problem Solving: Total System Intervention*. John Wiley. Chichester, UK. 1991.

[FJLP2001] Fabret F, Jacobsen HA, Llirbat F, Pereira J, Ross KA, and Shasha D. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. *Proceedings of the 2001 SIGMOD Conference*. May 2001.

[FK2001] Flores RA and Kremer RC. Bringing Coherence to Agent Conversations. In Woodridge M, Ciancarini P, and Weiss G (eds): *Agent-Oriented Software Engineering II*. Lecture Notes in Computer Science 2222. Springer-Verlag. Berlin, Germany. pp. 50-67. 2001.

[For82] Forgy CL. RETE: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence* . 19(1). pp. 17-37. September 1982.

[Fow98] Fowler M. Dealing with Roles. Supplement to *Analysis Patterns: reusable object models*. Addison-Wesley. Reading, MA. 1998.
Available at: http://www.aw.com/cseng/titles/0-201-89542-0/apsupp/roles2-1.html

[Fuc92] Fuchs NE. Specifications are (Preferably) Executable. *Software Engineering Journal*. 7(5). pp. 323-334. September 1992.

[FSS98] Fuchs NE, Schwertel U, and Schwitter R. Attempto Controlled English – Not Just Another Logic Specification Language. In P.Flener (ed.): *Logic-Based Program Synthesis and Transformation, Eighth International Workshop (LOPSTR '98)*. Manchester, UK. June 1998. Lecture Notes in Computer Science 1559. Springer-Verlag. Berlin, Germany. 1999.

[FSS99] Fuchs NE, Schwertel U, and Schwitter R. *Attempto Controlled English (ACE) Language Manual, Version 3.0*. Institut fur Informatik der Universitat Zurich. August 1999.

[GD93] Gatziu S and Dittrich KR. Events in an Active Object-Oriented Database System. *Proceedings of the 1st International Workshop on Rules in Database Systems*. Edinburgh, UK. August 1993.

[GD94] Gatziu S and Dittrich KR. Detective Composite Events in Active Database Systems Using Petri Nets. *Proceedings of the 4th International Workshop on Research Issues in Data Engineering: Active Database Systems.* Houston, Texas. February 1994.

[GJS92a] Gehani N, Jagadish HV, and Shmueli O. Composite event specification in active databases: Model and implementation. *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB'92).* Vancouver, Canada. August 1992.

[GJS92b] Gehani NH, Jagadish HV, and Shmueli O. Event specification in an active object-oriented database. *Proceedings of the International Conference on Management of Data (SIGMOD).* San Diego, CA. pp. 81-90. 1992.

[GLC99] Grosof BN, Labrou Y, and Chan HY. A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. In Wellman MP (ed.): *Proceedings 1st ACM Conference on Electronic Commerce (EC-99).* Denver, CO. ACM Press. New York, NY. November 1999.

[GM95] Gupta A, and Mumick IS. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering Bulletin.* 18(2). pp. 3-18. 1995.

[Goh97] Goh C. *A Generic Approach to Policy Description in System Management.* HP Laboratories Technical Report HPL-97-82. Bristol, UK. July 1997.

[GR2000] Green P and Rosemann M. Integrated Process Modelling: An Ontological Evaluation. *Information Systems.* 25(2). pp. 73-87. 2000.

[Gri94] Grishman R. *Computational Linguistics: An Introduction.* Cambridge University Press. Cambridge, UK. p. 97. 1994.

[GSSS2000] Greunz M, Schopp B, and Stanoevska-Slabeva K. Supporting Market Transactions through XML Contracting Containers. *Proceedings of the Sixth Americas Conference on Information Systems (AMCIS 2000).* Long Beach, CA. August 2000.

[GT98] Geppert A and Tombros D. Event-based Distributed Workflow Execution with EVE. In Davies N, Raymond K, and Seitz J (eds): *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing: Middleware '98.* Springer. 1998.

[Hal98] Halpin T. Object-Role Modelling (ORM/NIAM). Chapter 4 in: Bernus P, Mertins K, and Schmidt G (eds): *Handbook on Architectures of Information Systems*. Springer. 1998.

[Hal2001] Haley Enterprises. Eclipse and Café Rete. August 2001. http://www.haley.com/

[Han92] Hanson EN. Rule Condition Testing and Action Execution in Ariel. *Proceedings of the ACM SIGMOD Conference*. pp. 49-58. June 1992.

[Hay95] Hayes PJ. A Catalog of Temporal Theories. *Technical report UIUC-BI-AI-96-01*. Beckman Institute and Departments of Philosophy and Computer Science, University of Illinois. Chicago, IL. 1995.

[HBBM96] Hayton R, Bacon J, Bates J, and Moody K. Using Events to Build Large Scale Distributed Applications. *Proceedings of the ACM SIGOPS European Workshop '96*. Connemara, Ireland. 1996.

[Hay96] Hayton R. *OASIS: An Open Architecture for Secure Interworking Services*. PhD Thesis. University of Cambridge Computer Laboratory. March 1996.

[HBC97] Hanson EN, Bodagala S, and Chadaga U. Optimized Trigger Condition Testing in Ariel using Gator Networks. *Technical Report UF-CIS-TR-97-021*. CISE Department, University of Florida. Gainesville, FL. November 1997.

[HH97] Hay D and Healy KA. *GUIDE Business Rules Project. Final Report. Revision 1.2*. October 1997. Available from: http://www.businessrulesgroup.org/ Accessed on: 5 May 2000.

[HK93] Herrestad H and Krogh C. The Right Direction. *Towards a Global Expert System of Law*. Florence, Italy. MEDLAR II Deliverable V.2—3 243. December 1993.

[HKNPV98] Hanson EN, Konyala M, Noronha L, Park JB, and Vernon A. Scalable Trigger Processing in TriggerMan. *Technical Report 98-008*. University of Florida CISE Department. Gainesville, FL. July 1998.

[HM97] Hansson SO and Makinson D. Applying Normative Rules with Restraint. In Dalla ML (ed.): *Logic and Scientific Methods*. Kluwer. Dordrecht, The Netherlands. pp. 313-332. 1997.

[Hoh78] Hohfeld WN. *Fundamental Legal Conceptions as Applied in Judicial Reasoning*. Edited by Cook WW. Greenwood Press Publishers. Westport, CT. 1978.

[Hol92] Holland IM. Specifying reusable components using contracts. *Proceedings of the 6th European Conference on Object Oriented Programming (ECOOP'92)*. Springer-Verlag. Berlin. pp. 287-308. 1992.

[HPL99] Hoagland JA, Pandey R, and Levitt KN. Specifying and Enforcing Policies using LaSCO: the Language for Security Constraints on Objects. *Policy Workshop*. HP Laboratories, Bristol, UK. November 1999.

[HPV2000] Higginbotham J, Pianesi F, and Varzi A. *Speaking of Events*. Oxford University Press. Oxford, UK. 2000.

[HS98] Huhns MN and Singh MP. Agent Jurisprudence. *IEEE Internet Computing*. 2(2). pp. 90-91. March-April 1998.

[HV2001] Hitchens M and Varadharahan V. Tower: A Language for Role Based Access Control. In [SLL2001]. pp. 88-106.

[HvdVH97] Hoppenbrouwers J, van der Vos B, and Hoppenbrouwers S. NL Structures and Conceptual Modelling: Grammalizing for KISS. *Data and Knowledge Engineering*. 23(1). Special Issue: Natural Language for Data Bases (Workshop 1996). Elsevier Science Publishers. Amsterdam, The Netherlands. pp. 79-92. June 1997.

[IASC89] International Accounting Standards Committee. *Framework for the Preparation and Presentation of Financial Statements*. London, UK. 1989. http://www.iasc.org.uk/

[IASC97] International Accounting Standards Committee. *International Accounting Standards: IAS 33 (Earnings Per Share)*. London, UK. 1997. Available at: http://www.iasc.org.uk/

[IASC98] International Accounting Standards Committee. *International Accounting Standards: IAS 37 (Provisions, Contingent Liabilities, and Contingent Assets)*. London, UK. 1998. Available at: http://www.iasc.org.uk/

[IBM2001] IBM CommonRules. http://www.research.ibm.com/rules/home.html

[ILOG2001] ILOG Inc. ILOG Business Rule Components. August 2001. http://www.ilog.com/products/rules/rules_2001.pdf

[Ipe2001] Ipedo XML Data Management Platform. 2001. http://www.ipedo.com/

[ISO95] International Standards Organization (ISO/IEC JTC1/SC21/WG7). *Open Distributed Processing – Reference Model – Part 2: Foundations.* International Standard 10746-2 / ITU-T Recommendation X.902. 1995.

[ISO99a] International Standards Organization. *Database Language SQL.* Document ISO/IEC 9075. 1999.

[ISO99b] International Standards Organization. *Information Technology – Open Distributed Processing – Interface Definition Language.* Document ISO/IEC 14750. 1999.

[JFN2000] Jennings NR, Faratin P, Norman TJ, O'Brien P, and Odgers, B. Autonomous Agents for Business Process Management. *International Journal of Applied Artificial Intelligence.* 14(2). pp. 145-189. 2000.

[Jon2002] Jones AJI. *Conventional Signalling Acts.* Presented at the 24th Meeting of the Foundation for Intelligent Physical Agents (FIPA). Lausanne, Switzerland. 2002.

[JM2000] Jurafsky DS and Martin JH. *Speech and Language Processing.* Prentice Hall. New Jersey. pp. 499-543, 607-629. 2000.

[JS93] Jones AJI and Sergot M. On the Characterization of Law and Computer Systems: The Normative Systems Perspective. Chapter 12 in Meyer J-JCh and Wieringa RJ (eds): *Deontic Logic in Computer Science: Normative System Specification.* Kluwer Academic Publishers. 1993.

[JS96] Jones AJI and Sergot M. A formal characterisation of institutionalised power. *Journal of the Interest Group in Pure and Applied Logic.* 4(3). pp. 427-443. 1996.

[JS2000] Jones AJI and Sergot M. On Power in e-Institutions. *Second International Workshop on Formal Models of Electronic Commerce (FMEC'00).* Wharton School, University of Pennsylvania. Philadelphia, PA. 2000.

[JSS97] Jajodia S, Samarati P, and Subrahmanian VS. A Logical Language for Expressing Authorisations. *Proceedings of the IEEE Symposium on Security and Privacy.* Oakland, CA. pp. 164-174. May 1997.

[KCK2001] Kafeza E, Chiu DKW, and Kafeza I. View-Based Contracts in an E-Service Cross-Organizational Workflow Environment. *Proceedings of the Second International Workshop on Technologies for E-Services (TES'01).* Rome, Italy. Lecture Notes in Computer Science 2193. Springer-Verlag. Berlin, Germany. pp. 74-88. 2001.

[KGV99] Koetsier M, Grefen P, and Vonk J. *Cross-Organisational Workflow: CrossFlow ESPRIT E/28635* Contract Model, Deliverable D4b. 1999.

[Kim98a] Kimbrough SO. Sketch of a Basic Theory for a Formal Language for Business Communication. *Proceedings of the 31st Annual Hawaii International Conference on Systems Sciences (HICSS'98).* Kohalo Coast, Hawaii. January 1998.

[Kim98b] Kimbrough SO. On ESΘ Theory and the Logic of the X12 Date/Time Qualifiers. *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS98).* Kohalo Coast, Hawaii. IEEE Computer Society Press. pp. 176-185. January 1998.

[Kim2001] Kimbrough SO. Reasoning about the Objects of Attitudes and Operators: Towards a Disquotation Theory for the Representation of Propositional Content. *Eight International Conference on Artificial Intelligence and the Law (ICAIL 2001).* St Louis, MO. ACM Press. pp. 188-195. May 2001.

[KM93] Kimbrough SO and Moore SA. On Obligation, Time, and Defeasibility in Systems for Electronic Commerce. *Proceedings of the 26th Hawaii International Conference on Systems Sciences.* Kauai, Hawaii. IEEE Computer Society Press. pp. 493-502. January 1993.

[KM97] Kimbrough SO and Moore SA. On Automated Message Processing in Electronic Commerce and Work Support Systems: Speech Act Theory and Expressive Felicity. *ACM Transactions on Information Systems.* 15(4). ACM Press. New York, NY. pp. 321-367. October 1997.

[Koc97] Koch T. Automated Management of Distributed Systems. *Dissertation for the Degree of Doktor-Ingenieur at the Fachbereich fur Elektrotechnik der FernUniversitat.* Shaker Verlag. Hagen, Germany. 1997.

[KR93] Kamp H and Reyle U. *From Discourse to Logic.* Kluwer Academic Publishers. 1993.

[KR95] Krishnamurthy B and Rosenblum DS. Yeast: A General Purpose Event-Action System. *IEEE Transactions on Software Engineering*. 21(10). October 1995.

[Kra98] Kramer R. iContract – The Java Design By Contract Tool. *In Proceedings of the Technology of Object Oriented Languages and Systems (TOOLS26)*. IEEE. 1998.

[Kro97] Krogh C. Current Issues in the Field of Deontic Logic. *Doctor Philosophiae Lecture*. SINTEF Report. Oslo, Norway. June 1997.

[KRR97] Kappel G, Rausch-Schott S, and Retschitzegger W. *Implementing Business Rules on a Framework of Rule Patterns*. Technical Report 8/97. Department of Information Systems, University of Linz. 1997.

[KRR98] Kappel G, Rausch-Schott S, and Retschitzegger W. Coordination in Workflow Management Systems – A Rule-based Approach. In Conen W, Neumann G (eds): *Coordination Technology for Collaborative Applications – Organizations, Processes, and Agents*. Lecture Notes in Computer Science 1364. Springer-Verlag. Berlin, Germany. pp. 99-120. 1998.

[KRR2000] Kappel G, Rausch-Schott S, and Retschitzegger W: A framework for workflow management systems based on objects, rules and roles. *ACM Computing Surveys*. 32(1es). March 2000.

[KRRS2001] Kappel G, Rausch-Scott S, Retschitzegger W, and Sakkinen M. Bottom-up Design of Active Object-Oriented Databases. *Communications of the ACM*. 44(4). pp. 99-104. April 2001.

[KRRV94] Kappel G, Rausch-Schott S, Retschitzegger W, and Vieweg S. TriGS: Making a Passive Object-Oriented Database System Active. *Journal of Object-Oriented Programming*. 4. 1994.

[KS86] Kowalski R and Sergot M. A Logic-based Calculus of Events. *New Generation Computing*. 4. OHMSHA, LTD and Springer-Verlag. pp. 67-95. 1986.

[KT2000] Kimbrough SO and Tan Y-H. On Lean Messaging with Unfolding and Unwrapping for Electronic Commerce. *International Journal of Electronic Commerce*. 5(1). Fall 2000.

[Lam2001] van Lamsweerde A. Goal-Oriented Requirements Engineering: A Guided Tour. *5th IEEE International Symposium on Requirements Engineering (RE'01).* Toronto, Canada. pp. 249-263. August 2001.

[LBN99] Lobo J, Bhatia R, and Naqvi S. A Policy Description Language. *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-99).* MIT Press. Cambridge, MA. pp. 291-298. July 1999.

[LD92] Lee RM and Dewitz SD. Facilitating International Contracting: AI Extensions to EDI. *International Information Systems.* January 1992.

[Lee80] Lee RM. *CANDID: A Logical Calculus for Describing Financial Contracts.* PhD Thesis. Department of Decision Sciences (now Department of Operations and Information Management), The Wharton School, University of Pennsylvania. Philadelphia, PA. June 1980.

[Lee88] Lee RM. Bureaucracies as Deontic Systems. *ACM Transactions on Office Information Systems* 6(2). pp. 87-108. April 1988.

[Lee98] Lee RM. Towards Open Electronic Contracting. *Journal of Electronic Markets, Special Issue on Electronic Contracting.* 8(3). pp. 3-8. 1998.

[Lin77] Lindahl L. *Position and Change – A Study in Law and Logic.* D Reidel Publishing Company. Dordrecht, The Netherlands. 1977.

[Liu2000] Liu K. *Semiotics in Information Systems Engineering.* Cambridge University Press. 2000.

[Llo2000] Lloyd MS. *Conversion of Access Control Policy to Formal Logic.* MPhil Thesis. University of Cambridge Computer Laboratory. 2000.

[LO99] Liu K and Ong T. A Modelling Approach for Handling Business Rules and Exceptions. *The Computer Journal.* 42(3). pp. 221-231. 1999.

[LSDM2001] Liu K, Sun L, Dix A, and Mohan N. Norm Based Agency for Designing Collaborative Systems. *Information Systems Journal.* 11(3). Blackwell Science. Oxford, UK. pp. 229-247. 2001.

[LPT99] Liu L, Pu C, and Tang W. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering.* 11(4). July/August 1999.

[LR94] Lee RM and Ryu YU. DX: A Deontic Expert System. *Journal of Management Information Systems*. 12(1). pp. 145-169. 1995.

[LS97] Lupu E and Sloman M. A Policy Based Role Object Model. *Proceedings of the 1st IEEE Enterprise Distributed Object Computing Workshop (EDOC'97).* Gold Coast, Australia. pp. 36-47. October 1997.

[LS99] Lupu E and Sloman M. Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering.* Special Issue on Inconsistency Management. 25(6). pp. 852-869. November/December 1999.

[Mak86] Makinson D. On the Formal Representation of Rights Relations. *Journal of Philosophical Logic*. 15. pp. 403-425. 1986.

[Mak88] Makinson D. Rights of Peoples: Point of View of a Logician. In Crawford J (ed.): *The Rights of Peoples*. Oxford University Press. Oxford, UK. pp. 69-92. 1988.

[Mak99] Makinson D. On a Fundamental Problem of Deontic Logic. In McNamara P and Prakken H (eds): *Norms, Logics and Information Systems. New Studies in Deontic Logic and Computer Science.* IOS Press. Amsterdam, The Netherlands. pp. 29-53. 1999.

[MB98] Ma C and Bacon J. COBEA: A CORBA-Based Event Architecture. *Proceedings USENIX COOTS'98*. Santa Fe, NM. pp. 117-131. April 1998.

[MBBR95] Milosevic Z, Berry A, Bond A, and Raymond K. Supporting Business Contracts in Open Distributed Systems. *2nd International Workshop on Services in Distributed and Networked Environments (SDNE'95).* Whistler, Canada. June 1995.

[MC92] Masullo MJ and Calo SB. *Policy Management: An architecture and approach.* Research Report RC 18505 (80943) 11/10/92. IBM Research Division. November 1992.

[MDBS69] Martin RM, Davidson D, Butler RJ, and Salmon WC. Proceedings of the Symposium on Events and Event-Descriptions at the University of Western Ontario Philosophy Colloquium 1966. In Margolis J (ed.): *Fact and Existence.* Basil Blackwell. Oxford, UK. 1969.

[Mer98] Merz M. Electronic Contracting with COSMOS – How to Establish, Negotiate, and Execute Contracts on the Internet. In Kobryn C, Atkinson C, and Milosevic Z (eds): *Proceedings of the 2nd International Enterprise Distributed Object Computing Workshop (EDOC'98).* IEEE. November 1998.

[MESW2001] Moore B, Elleson E, Strassner J, and Westerinen A. *Policy Core Information Model – Version 1 Specification.* IETF RFC 3060. 2001.
Available at: http://www.ietf.org/rfc/rfc3060.txt?number=3060

[Mey97] Meyer B. *Object-Oriented Software Construction.* 2nd Edition. Prentice Hall. New Jersey. 1997.

[Mey99] Meyer B. *Building Bug-Free OO Software: An Introduction to Design By Contract.* Available at: http://www.eiffel.com/doc/manuals/technology/contract/

[MGTM98] Merz M, Griffel E, Tu T, Muller-Wilken S, Weinreich H, Boger M, and Lamersdorf W. Supporting Electronic Commerce Transactions with Contracting Services. *International Journal of Cooperative Information Systems.* 7(4). World Scientific Publishing Company. pp. 249-274. December 1998.

[Mic2001a] Microsoft Site Server. http://www.microsoft.com/siteserver/. 2001.

[Mic2001b] Microsoft Commerce Server. http://www.microsoft.com/commerceserver/. 2001.

[Mil95] Milosevic Z. *Enterprise Aspects of Open Distributed Systems.* PhD Thesis. Department of Computer Science, University of Queensland. pp. 154-248. October 1995.

[Mir87] Miranker DP. TREAT: A better match algorithm for AI production systems. *Proceedings Sixth National Conference on Artificial Intelligence (AAAI-87).* Morgan Kaufmann. San Francisco, CA. August 1987.

[Moo2000] Moore SA. KQML and FLBC: Contrasting Agent Communication Languages. *International Journal of Electronic Commerce.* 5(1). Fall 2000.

[MOR2001] Michael JB, Ong VL, and Rowe NC. Natural Language Processing Support for Developing Policy-Governed Software Systems. *39th International Conference on Object-Oriented Languages and Systems (TOOLS USA 2001).* Santa Barbara, CA. July-August 2001.

[Mou78] Mourelatos APD. Events, Processes, and States. *Linguistics and Philosophy*. 2. pp. 415-434. 1978. Reprinted in P. Tedeschi and A Zaenen (eds): Syntax and Semantics 14: Tense and Aspect. Academic Press. New York, NY. pp. 191-212. 1981.

[MS93] Moffett JD and Sloman MS. Policy Hierarchies for Distributed Systems Management. *IEEE Journal on Selected Areas in Communications*. 11(9). pp. 1404-1414. December 1993.

[MS94] Moffett JD and Sloman MS. Policy Conflict Analysis in Distributed System Management. *Journal of Organizational Computing*. 4(1). Ablex Publishing. pp. 1-22. 1994.

[MS97] Mansouri-Samani M and Sloman M. GEM: A Generalized Event Monitoring Language for Distributed Systems. *IEEE/IOP/BCS Distributed Systems Engineering Journal*. 4(2). June 1997.

[MSM2001] Morciniec M, Salle M, and Monahan B. Towards Regulating Electronic Communities with Contracts. *2nd Workshop on Norms and Institutions in Multi-Agent Systems, at the 5th International Conference on Autonomous Agents*. Montreal, Canada. May 2001. Also available as Hewlett Packard Technical Report HPL-2001-120. Available at: http://www.hpl.hp.com/techreports/2001/HPL-2001-120.pdf

[MSY95] Marriott DA, Sloman M, and Yialelis N. *Management Policy Service for Distributed Systems*. Research Report DoC 95/10. Department of Computing, Imperial College. London, UK. October 1995.

[MU2000] Minsky NH and Ungureanu V. Law-Governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems. *ACM Transactions on Software Engineering and Methodology*. 9(3). pp. 273-305. 2000.

[MvdT2001] Makinson D and van der Torre L. Constraints for Input/Output Logics. *Journal of Philosophical Logic*. 30. pp. 155-185. 2001.

[MW93] Meyer JJ and Wieringa RJ. *Deontic Logic in Computer Science: Normative System Specification*. John Wiley and Sons. Chichester, UK. 1993.

[NSJ98] Norman TJ, Sierra C, and Jennings NR. Rights and commitments in multi-agent agreements. *Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS-98)*. Paris, France. pp. 222-229. 1998.

[OAS2001] Organization for the Advancement of Structured Information Standards (OASIS). *An Introduction to the Provisioning Services Technical Committee*. 2001. Available at: http://www.oasis-open.org/committees/provision/Intro-102301.doc

[OAS2002] Organization for the Advancement of Structured Information Standards (OASIS). *OASIS ebXML Collaboration-Protocol Profile and Agreement Specification Version 2.0*. June 2002. Available at: http://www.oasis-open.org/committees/ebxml-cppa/documents/ebcpp-2.0.pdf

[OM96] Osborne M and MacNish C. Processing natural language software requirements specifications. *International Conference on Requirements Engineering*. Colorado Springs, CO. 1996.

[OMG2001] Object Management Group. OMG Unified Modelling Language Specification, Version 1.4. September 2001. Available at: http://www.omg.org/technology/documents/formal/uml.htm

[Par90] Parsons T. *Events in the Semantics of English: A Study in Subatomic Semantics*. MIT Press. Cambridge, MA. 1990.

[PD99] Paton NW and Diaz O. Active Database Systems. *ACM Computing Surveys*. 31(1). pp. 63-103. 1999.

[PES2000] Peyton Jones S, Eber JM, and Seward J. Composing contracts: An adventure in financial engineering. *International Conference on Functional Programming*. Montreal, Canada. September 2000.

[PL2000] Pottinger R and Levy A. A Scalable Algorithm for Answering Queries using Views. *Proceedings of the 26th International Conference on Very Large Databases (VLDB 2000)*. pp. 484-495. 2000.

[PS97] Prakken H and Sergot M. Dyadic Deontic Logic and Contrary-to-Duty Obligations. In Nute D (ed.): *Defeasible Deontic Logic: Essays in Nonmonotonic Normative Reasoning*. Synthese Library No. 263. Kluwer Academic Publishers. pp. 223-262. 1997.

[PS98] Patankar AK and Segev A. An Architecture and Construction of a Business Event Manager. In Etzion O, Jajodia S, and Sripada S (eds): *Temporal Databases – Research and Practice*. Lecture Notes in Computer Science 1399. Springer-Verlag. Berlin, Germany. pp. 257-280. 1998.

[PST2001] Production Systems Technologies Inc. Rete and Rete II. August 2001. http://www.pst.com/rete.htm

[Pul96] Pulman SG. Controlled Language for Knowledge Representation. *Proceedings of the First International Workshop on Controlled Language Applications (CLAW96)*. Katholieke Universiteit Leuven. Belgium. pp. 233-242. March 1996.

[RCS83] Robinson RE, Coval SC, and Smith JC. The Logic of Rights. *University of Toronto Law Journal*. 33(267). 1983.

[Ret98] Retschitzegger W. Composite Event Management in TriGS – Concepts and Implementation. In Quirchmayr G, Schweighofer E, Bench-Capon TJM (eds): *Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA '98)*. Vienna, Austria. August 1998. Lecture Notes in Computer Science 1460. Springer-Verlag. Berlin, Germany. 1998.

[RGW2002] Reeves DM, Grosof BN, and Wellman MP. Automated Negotiation from Declarative Contract Descriptions. *Computational Intelligence*. Special Issue on Agent Technologies for Electronic Commerce. Forthcoming, 2002.

[Ril2001] Riley G. CLIPS: A Tool for Building Expert Systems. August 2001. Available at: http://www.ghg.net/clips/CLIPS.html and http://www.ghgcorp.com/clips/CLIPS-FAQ

[RZF2001] Ribeiro C, Zúquete A, and Ferreira P. Enforcing Obligation with Security Monitors. *Proceedings of the Third International Conference on Information and Communications Security (ICICS2001)*. Xian, China. Lecture Notes in Computer Science 2229. Springer-Verlag. Berlin, Germany. pp. 172-176. November 2001.

[SABH2000] Segall B, Arnold D, Boot J, Henderson M, and Phelps T. Content Based Routing with Elvin4. *Proceedings AUUG2K*. Canberra, Australia. June 2000. Available at: http://www.elvin.dstc.edu.au/doc/papers/index.html

[SB98] Spiteri M and Bates J. An Architecture for the Storage and Retrieval of Events. *Proceedings of Middleware'98*. Springer. pp. 443–458. 1998.

[SC96] Santos F and Carmo J. Indirect action, influence, and responsibility. In Brown M and Carmo J (eds): *Deontic Logic, Agency, and Normative Systems*. Springer. pp. 194-215. 1996.

[Sch96] Schwiderski S. *Monitoring the Behaviour of Distributed Systems*. PhD Thesis. University of Cambridge Computer Laboratory. April 1996.

[Sch99] Schmidt M. The Evolution of Workflow Standards. *IEEE Concurrency*. July-September 1999.

[SCJ97] Santos F, Carmo J, and Jones AJI. Action concepts for describing organized interaction. In Sprague RA (ed.): *30th Annual Hawaii International Conference on Systems Sciences*. pp. 373-382. 1997.

[SD2000] Steen MWA and Derrick J. ODP Enterprise Viewpoint Specification. *Computer Standards and Interfaces*. 22. pp. 165-189. September 2000.

[Sea69] Searle JR. *Speech acts: An essay in the philosophy of language*. Cambridge University Press. 1969.

[Ser99] Sergot M. Deontic Logic in Policy Specification. *Policy Workshop*. HP Laboratories, Bristol, UK. November 1999.

[Ser2001] Sergot M. A Computational Theory of Normative Positions. *ACM Transactions on Computational Logic*. 2(4). pp. 581-622. October 2001.

[SGTV2000] Schlenoff C, Gruniger M, Tissot F, Valois J, Lubell J, and Lee J. The Process Specification Language (PSL) Overview and Version 1.0 Specification. *NISTIR 6459*. National Institute of Standards and Technology (NIST). Gaithersburg, MD. 2000.

[SHP88] Stonebraker M, Hanson EN, and Potamianos S. The POSTGRES Rule Manager. *Transactions on Software Engineering*. 14(7). July 1988.

[SIL2002] SIL International. *What is discourse deixis?* July 2002. Available at: http://www.sil.org/linguistics/GlossaryOfLinguisticTerms/WhatIsDiscourseDeixis.htm

[SK91] Stonebraker M and Kemnitz G. The Postgres Next-Generation Database Management System. *Communications of the ACM*. 34(10). October 1991.

[SK95] Sadri F and Kowalski R. Variants of the Event Calculus. *In Proceedings of the 12th International Conference on Logic Programming (ICLP 95)*. MIT Press. Cambridge, MA. pp. 67-82. 1995.

[SLL2001] Sloman M, Lobo J, and Lupu EC (eds.): *Proceedings of Policies for Distributed Systems and Networks: International Workshop (Policy 2001).* Bristol, UK. Lecture Notes in Computer Science 1995. Springer-Verlag. Berlin, Germany. January 2001.

[Slo94] Sloman M. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management.* 2(4). Plenum Press. 1994.

[Slo95] Sloman M. Management Issues for Distributed Services. *Proceedings IEEE 2nd International Workshop on Services in Distributed and Networked Environments (SDNE95).* Whistler, BC. June 1995.

[Slo2000] Sloman M, Dulay N, and Nuseibeh B. *EPSRC Grant GR/1 96103. SecPol: Specification and Analysis of Security Policy for Distributed Systems.* Imperial College, Department of Computing. 2000.

[SLR88] Sellis T, Lin C-C, and Raschid L. Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms. *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data.* Chicago, IL. pp. 404-412. June 1988.

[Smi80] Smith R. The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers.* 29(12). December 1980.

[Sof2001] Software AG Tamino XML Server. http://www.softwareag.com/tamino/. 2001.

[Sow2000] Sowa JF. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* Brooks/Cole. Pacific Grove, CA. 2000.

[Spi2000] Spiteri MD. *An Architecture for the Notification, Storage, and Retrieval of Events.* PhD Thesis. University of Cambridge Computer Laboratory. January 2000.

[SSKK86] Sergot MJ, Sadri F, Kowalski RA, Kriwaczek F, Hammond P, and Cory HT. The British Nationality Act as a Logic Program. *Communications of the ACM.* 29(5). pp. 370-386. May 1986.

[Sta2002] Staskiewicz D. *Logikbasierte Überwachung der Vertragserfüllungsphase im elektronischen Handel.* Diplomarbeit Informatik am Lehrstuhl IV der Rheinisch-Westfälische Technische Hochschule Aachen (RWTH Aachen). Aachen, Germany. December 2002. Forthcoming.

[Ste2000] Steedman M. *The Productions of Time.* Tutorial notes, Draft 4.0. Cognitive Science, Division of Informations, University of Edinburgh. July 2000.

[SV85] Searle JR and Vanderveken D. *Foundations of Illocutionary Logic.* Cambridge University Press. Cambridge, UK. 1985.

[SvdAA99] Sheth AP, van der Aalst WMP, and Arpinar IB. Processes Driving the Networked Economy. *IEEE Concurrency.* July-September 1999.

[SvdRD2000] Steuten AAG, van de Riet RP, and Dietz JLG. Linguistically based conceptual modelling of business communication. *Data and Knowledge Engineering.* 35. pp. 121-136. 2000.

[TB99] Thorpe CP and Bailey JCL. *Commercial Contracts.* Kogan Page Limited. London, UK. 1999.

[TGNO92] Terry D, Goldberg D, Nichols D, and Oki B. Continuous Queries over Append-Only Database. *Proceedings of the 1992 ACM SIGMOD International Conference on the Management of Data.* San Diego, CA. pp. 321-330. June 1992.

[Tho98] Thomas J. *Meaning in Interaction: An Introduction to Pragmatics.* Addison Wesley Longman Limited. New York, NY. pp. 1-54. 1998.

[Tom99] Tombros D. *An Event- and Repository-based Component Framework for Workflow System Architecture.* PhD Thesis. University of Zurich. 1999.

[TT99] Tan Y-H and Thoen W. A Logical Model of Directed Obligations and Permissions to Support Electronic Contracting in Electronic Commerce. *International Journal of Electronic Commerce (IJEC).* 3(2). pp. 87-104. Winter 1998-1999.

[TW2001a] Taveter K and Wagner G. Agent-Oriented Business Rules: Deontic Assignments. *In Proceedings of the International Workshop on Open Enterprise Solutions: Systems, Experiences, and Organizations (OES-SEO2001).* Rome, Italy. September 2001.

[TW2001b] Taveter K and Wagner G. Agent-Oriented Enterprise Modelling Based on Business Rules. *In Proceedings of the 20th International Conference on Conceptual Modelling (ER2001).* Yokohama, Japan. November 2001.

[UM96] University of Michigan. *The SLAPD and SLURPD Administrator's Guide: Release 3.3.* April 1996.

[Uso2001] Usoft Developer Series. http://www.ness-europe.com/products/usoftdeveloperseries/default.htm. 2001.

[vdA96] van der Aalst WMP. Three Good Reasons for Using a Petri-Net-based Workflow Management System. In Navathe S and Wakayama T (eds): *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96).* pp. 179-201. Cambridge, MA. November 1996.

[vdAvHH94] van der Aalst WMP, van Hee KM, and Houben GJ. Modelling and analysing workflow using a Petri-Net-based approach. In De Michelis G, Ellis C, and Memmi G (eds): *Proceedings of the Second Workshop on Computer-Supported Cooperative Work, Petri Nets and Related Formalisms.* pp. 31-50. 1994.

[Ver96] Verkuyl HJ. *A Theory of Aspectuality.* Cambridge University Press. 1996.

[Ver2001] Versata Logic Suite. http://www.versata.com/. 2001.

[vW51] von Wright GH. Deontic Logic. *Mind.* 60. pp. 1-15. 1951.

[Wag2000] Wagner G. Agent-Oriented Analysis and Design of Organizational Information Systems. In J. Barzdins and A. Caplinskas (eds): *Databases and Information Systems: 4th IEEE International Baltic Workshop on Databases and Information Systems.* Vilnius, Lithuania. May 2000.

[Wag2001] Wagner G. The Agent-Object-Relationship Meta-Model: Towards a Unified View of State and Behaviour. *Technical Report.* Eindhoven University of Technology. August 2001.

[WfMC95] Workflow Management Coalition (WfMC). The Workflow Reference Model: Issue 1.1. *TC00-1003.* 1995. Available at: http://www.wfmc.org/standards/docs.htm

[Wid96] Widom J. The Starburst Active Database Rule System. *IEEE Transactions on Knowledge and Data Engineering.* 8(4). pp. 583-595. 1996.

[Wies95] Wies R. Using a Classification of Management Policies for Policy Specification and Policy Transformation. *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*. Santa Barbara, CA. May 1995.

[WX2001] Weigand H and Xu L. Contracts in E-Commerce. *9th IFIP 2.6 Working Conference on Database Semantic Issues in E-Commerce Systems (DS-9)*. April 2001.

[[[[This page exists just to force the printer beech to print the last page of the bibliography]]]]

[[[[This page exists just to force the printer beech to print the last page of the bibliography]]]]

[[[[This page exists just to force the printer beech to print the last page of the bibliography]]]]

[[[[This page exists just to force the printer beech to print the last page of the bibliography]]]]