Mechanical Consistency Analysis for Business Contracts and Policies

Alan S. Abrahams, David M. Eyers, and Jean M. Bacon

University of Cambridge
Computer Laboratory
Cambridge, Cambridgeshire, United Kingdom
{Alan.Abrahams,David.Eyers,Jean.Bacon}@cl.cam.ac.uk

Abstract

The EDEE system provides a framework through which businesses may store the data pertaining to business events, contracts and organizational policies, within a single repository using the unifying notion of an occurrence. A collection of stored queries (cf. SQL views) is maintained. Each query describes the desirable, undesirable, and not-undesirable occurrences under the provisions (obligations, prohibitions, and privileges or immunities) of the contracts and policies of an organization. Desirable occurrences are those that are obliged according to some *obligation*. Undesirable occurrences are those that lead to violation, as per some explicit or implicit *prohibition*. Not-undesirable occurrences are those that explicitly do not lead to violations, as specified by some *privilege or immunity*. This paper proposes a mechanism for the dynamic derivation of the overlaps between provisions. We show, through a worked example, that by determining the covering relationships between stored queries we can mechanically discover inconsistencies in business contracts and organizational policies between obligations and privileges, and between different descriptions of obligations.

Keywords occurrence, contracts, policies, conflict detection, conflict resolution.

1 INTRODUCTION

Prudent business enterprises operating in e-service environments need to check proposed business contracts against their organizational rules, to ensure that their intentions do not violate internal regulations. The E-commerce application Development and Execution Environment, or EDEE, system [3] unifies storage of data pertaining to real business events, prospective actions and business policy through the notion of occurrences and queries over these occurrences.

In this paper, we propose a framework for storing contracts and policies, and for checking their consistency. We view both contracts and policies as sets of provisions. A *provision* specifies an obligation, prohibition, privilege (immunity), or power. In the case of obligations, prohibitions, and privileges, each provision embeds a query which describes, respectively, the desirable, undesirable, or not-undesirable occurrences. For powers, the provision describes which conventional occurrences bring about legal relations, such as additional obligations, prohibitions, and privileges. Our system facilitates dynamic addition of provisions, and through automatic derivation of overlaps between stored queries, can ascertain conflicts.

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 chief clerk, John James, to the database, the query 'deliveries to **the chief**

clerk' is dirtied as the results must now also include any 'deliveries to **John James**'. Any such deliveries would be what we term *output dirt*. The materialized view literature [9] talks of dirt in the sense of our output dirt. Whereas materialized views would only change when any actual deliveries to John James were added, covering relationships may change even in the absence of any deliveries stored in the database: queries may come into overlap, or become partially or completely disjoint as new data is added. As we will demonstrate, changes in the covering relationships may bring obligations for a particular debt stored in the database into conflict.

This paper shows how conflicts may be detected at the time contracts are added to the database, or when inserted data brings provisions into conflict. The examples we present show how we discover two major types of conflict: conflicts between obligations and privileges, and conflicts between different descriptions of what occurrences are required to fulfill an obligation. The first type of conflict is demonstrated in an example which depicts a conflict between a obligation to pay within a certain time, and a privilege (derived from an obligation of lesser strictness) to pay within a more lenient time-scale. The second type of conflict – inconsistent descriptions of how to service a particular debt – are illustrated for various example cases: obligations specifying different times to pay, and obligations specifying different people to deliver to.

In many cases conflicts are unavoidable and there is no possibility of satisfying both obligations. However, some conflicts are less problematic in that the conflict is a only a potential one, and it is possible to fulfill both 'conflicting' obligations. Nevertheless, even for this weaker case, companies often prefers to remove the potential conflict entirely, to avoid possible future disputes. Conflict can be removed by violating or voiding the source obligations that lead to the conflict. Conflict resolution is treated in [2]; this paper concentrates only on conflict detection. The new conflict detection cases illustrated here supplement the earlier cases dealt with in [1].

We begin with a review of related work (Section 2). Then, via an application scenario (Section 3) we describe how operational data, specific and general provisions, and queries may be stored. We illustrate how inconsistent provisions can be found (Section 4) and look at the times at which potential and actual conflicts can be detected (Section 5).

2 RELATED WORK

Previous contract assessment approaches, such as [5, 7], apply Petri Nets or Finite State machines to determine contract status. Contracts are reduced to directed graphs that capture the business procedure, but leave provisions implicit. To allow inspection and analysis, provisions need to be explicitly captured within the business database. Explicit storage of provisions can then be exploited for consistency checking, contract performance assessment, and management review of which provisions pertain to items or occurrences.

It is instructive to contrast EDEE with traditional expert systems approaches to business logic. The occurrence database is the working memory of the system; production rules in EDEE are maintained in a list of queries over this database. These queries are explicitly stored criteria describing sets of items and occurrences. As such, they are more similar to SQL views, than to the throw-away queries executed by an SQL engine.

The EDEEQL extension of SQL [3] leads to an occurrence structure with a simple tabular form able to store business events and provisions of contracts and policies. It avoids the need to specify schemas explicitly for each occurrence class, thus increasing the dynamic configurability of the system. This particular storage approach has been chosen for semantic rather than performance reasons. The representation allows us to determine when parties participate in the same occurrence, but unlike full graph-based representations (for example, the Hydra database system [4]), we cannot directly locate more distant associations.

An underlying database system manages storage and retrieval of occurrences and queries. The coverage checking mechanism proposed in this paper optimizes the execution of these stored queries. Due to the common goal of incremental state re-computation, it has many similarities to the RETE [8] and TREAT [17] expert system optimization algorithms. The most striking difference is that our approach places an emphasis on dynamic compilation and analysis of coverage. This allows us to go beyond the fact/pattern (object/query) matching in RETE and TREAT to also perform pattern/pattern (query/query) matches as well.

Table 1: A tabular schema for storing various occurrences

Commentary	Occurrence	Role	Participant
John James *being the chief clerk* for SkyHi	being_chief_clerk1	chief_clerk	John James
		occurred_on	1 Aug 2002
SkyHi *ordering* the consignment	ordering1	orderer	SkyHi
		ordered	consignment1
Steelmans *delivered* the consignment to SkyHi on 1 September 2002	delivering1	deliverer	Steelmans
		delivered	consignment1
		recipient	SkyHi
		occurred_on	1 Sep 2002

3 APPLICATION SCENARIO

We introduce an application scenario, describe how operational data, provisions, and queries are stored, then illustrate via worked examples how conflicts between a contract and a company's standard terms and conditions are determined.

In our scenario, 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, whereby Steelmans is to deliver a consignment (say, `consignment1`) of steel. At the time of entering the contract John James is the chief clerk. We select some hypothetical clauses from this contract: **Clause C.1** SkyHi is obliged to pay Steelmans before 10 September 2002 and **Clause C.2** Steelmans must deliver the consignment to John James. We also take two clauses from the January 2001 version of SkyHi's standard terms and conditions (i.e. general business policies): **Clause P.3** SkyHi is obliged to pay suppliers within 20 days of delivery, and **Clause P.4** All orders must be delivered to the chief clerk.

3.1 Storing Operational Data

Let us say John James is chief clerk, SkyHi has ordered the consignment, and, Steelmans, the supplier, has delivered the consignment to SkyHi. Let `being_chief_clerk1`, `ordering1`, and `delivering1` denote instances (hence the `1` added to create a unique identifier) of occurrences of type *being a chief clerk*, *ordering*, and *delivering* respectively. Table 1 shows the occurrence, role, participant schema employed in EDEE to store this operational data. For readability we have included values like `John James`, `SkyHi`, and `Steelmans` in our tables instead of foreign key references. Similarly we show occurrence primary keys in forms such as `being_chief_clerk1`, instead of foreign key references into a table describing the occurrence type (`being_chief_clerk`). Finally we omit repeated key values in adjacent rows.

3.2 Storing Specific and General Provisions of Contracts and Policies

To store contractual provisions – e.g. "it is *obliged* that [X pay Y before date Z]", "it is *prohibited* that [X pay Y before date Z]", "it is *permitted* that [X pay Y after date Z]" – in a relational database we need to handle their embedded propositional content [3, 12]. This section demonstrates the storage of obligations and privileges; the storage of prohibitions is dealt with elsewhere [1].

3.2.1 Storing Specific and General Obligations

Consider **Clause C.1** from the application scenario presented above, which encodes a *specific* obligation of SkyHi to pay Steelmans before 10 September 2002. Clearly, to store the obligation, we cannot simply store an occurrence, say `paying1`,

Table 2: Schema for storing a specific obligation

Occurrence	Role	Participant
being_obliged1	obliged	query9
	purpose	consignment1
	isAccordingTo	ClauseC1
being_obliged2	obliged	query17
	purpose	ordering1
	isAccordingTo	ClauseC2

(query9 = first occurrence of paying Steelmans before 10 September 2002. See Figure 1)

(query17 = first occurrence of delivering to John James. See Figure 3)

of "SkyHi paying Steelmans before 10 September 2002" because paying1 is a concrete instance and might not yet have occurred anyway. We instead store the obligation as being_obliged1 in Table 2, and indicate the obliged occurrences using a pointer to a database view (query) describing the set of obliged (desirable) occurrences, which is query9 in Figure 1. query9 asks for the *first* payment, since it is exactly one payment that is promised. The query may be empty in cases where the obligation has not been fulfilled, and no payments fitting the description have been made [1].

In a similar manner, we can capture the specific obligation encoded in **Clause C.2** as being_obliged2 in Table 2. being_obliged2 is an obligation to deliver to John James, pertaining to the order ordering1, and arising from **Clause C.2**.

The purpose role of each obligation associates the obligation with its purpose: that is, specifies what the performance mandated by the obligation is *for*. In the case of the obligation in **Clause C.1** of our application scenario, the obligation is an obligation to pay *for* consignment1; in **Clause C.2** the obligation is to deliver *for* ordering1. The purpose role is roughly comparable to the $sake(\ldots)$ predicate employed by Kimbrough [12].

The isAccordingTo role of each obligation captures the notion that each obligation is a *prima facie* obligation that exists according to a particular textual or verbal utterance that is contained in some identified document or discourse. We say that obligations are *prima facie* because they may be voided by other clauses (see [2]).

To encode a *general* obligation policy, such as that defined in **Clause P.3** of our obligation scenario we define a function. A function takes as its domain a set of occurrences of a given description, and produces, in its range, occurrences of a particular form. To capture **Clause P.3**, we might have:

being_obliged_function1:
occurrences of a supplier *delivering* →
occurrences of SkyHi *being_obliged* to pay within 20 days of the occurrence of delivering

That is to say, being_obliged_function1, maps occurrences of delivering to occurrences of being obliged. In tabular form, we may record the rule that, according to **Clause P.3** of SkyHi's standard terms and conditions, deliveries bring about (prima facie) obligations to pay within 20 days, as shown in Table 3 [2]. To capture the fact that these rules (functions) are subjective determinations of legal consequence made by particular laws, we use an isAccordingTo attribute to store the clause identifier for each rule, and we capture also the utterer, or provenance in a document, of the clause. The occurrence being_in1 denotes that **Clause P.3** is contained in a particular version of the standard terms and conditions for SkyHi.

Taking the delivery, delivering1, made by Steelmans to SkyHi on 1 September this yields the following identified obligation to pay, originating from the application of being_obliged_function1 (in the standard terms and condi-

[1] Technically, to ensure that a given payment satisfies only one obligation we should allocate payments to obligations, as shown in [3]. However, as this is not pertinent to our current discussion, we omit that complication here.

[2] The notation |queryX| denotes a bound variable: merely substitute the item covered by queryX for |queryX|. For example, delivering1 is covered by query23, thus we transform |query23| to delivering1

Table 3: Representing the general rule that a delivery brings about an obligation to pay within 20 days of delivery

Occurrence	**Role**	**Participant**
being_obliged_function1	domain	query23
	isAccordingTo	clauseP3
	obliged	query24
being_in1	contents	clauseP1
	container	standard_terms_version1
	purpose	query27

(query23 = occurrences of [delivering])

(query24 = first of [occurrences of [paying] where [< [participants in role [=occurred_on] in |query23|] + 20 days] is [=occurred_on]] in [ascending] [temporal] order)

(query27 = [participants in role [=delivered] in |query23|])

Table 4: Representing the general rule that an order brings about an obligation to deliver that order to the chief clerk

<table>
<tr><th>Occurrence</th><th>Role</th><th>Participant</th></tr>
<tr><td>being_obliged_function2</td><td>domain</td><td>query25</td></tr>
<tr><td></td><td>isAccordingTo</td><td>clauseP4</td></tr>
<tr><td></td><td>obliged</td><td>query22</td></tr>
<tr><td></td><td>purpose</td><td>|query25|</td></tr>
<tr><td>being_in2</td><td>contents</td><td>clauseP4</td></tr>
<tr><td></td><td>container</td><td>standard_terms_version1</td></tr>
</table>

(query25 = occurrences of [ordering])

(query22 = first of [occurrences of [delivering] where [participant in role [=chief_clerk] in [occurrences of [being_chief_clerk]]] are [=recipient]]). See Figure 4)

tions) to delivering1:

```
being_obliged3 =                                                              (legal consequence)
  source_rule       : being_obliged_function1                                        (provenance)
  source_occurrence : delivering1                                                      (evidence)
  obliged           : query12 (first occurrence of paying by 21 Sep 2002; see Figure 2)
  purpose           : consignment1
```

being_obliged3 (where 3 has been chosen as a unique identifier) captures the particular legal consequence brought about from applying rules of exact provenance, to specific evidence. The link between a rule (function), a happening, and the resultant conclusions is recorded by storing the sources of the conclusion as its source_rule and source_occurrence attributes. Note that query12 (which is first of [occurrences of [paying] where [<21 Sep 2002]] is [=occurred_on]] in [ascending] [temporal] order) was obtained by substituting delivering1 for |query23| in the parameterized query, query24 as introduced in Table 3, and resolving [participants in role [=occurred_on] in [=delivering1] + 20 days] to get 21 Sep 2002. Further, consignment1 in role purpose was obtained by substituting delivering1 for |query23| in the parameterized query, query27, to obtain [participants in role [=delivered] in [=delivering1]], which resolves to consignment1.

The general obligation policy, from **Clause P.4**, that all orders be delivered to the chief clerk, can be depicted as being_obliged_function2, as shown in Table 4. As can be seen, **Clause P.4** maps occurrences of ordering to obligations to deliver the order to the chief clerk.

For the order, ordering1, then, we can deduce the following specific obligation arising from **Clause P.4** of the

Table 5: Capturing the rule that not paying for the consignment before 10 September leads to violation

Occurrence	**Role**	**Participant**
`violating_function1`	`domain`	`query28`
	`violated`	`being_obliged1`

(`query28 = occurrences of [counting] where [>=10 September] is [=occurred_on] intersection occurrences of [counting] where [0] is [=count] intersection occurrences of [counting] where [query9] is [=counted]`).

(`query9` is embedded in the obligation `being_obliged1` of Table 2, and defined in Figure 1).

standard terms and conditions:

```
being_obliged4 =                                               (legal consequence)
  source_rule : being_obliged_function2                        (provenance)
  source_occurrence : ordering1                                (evidence)
  obliged           : query22 (first occurrence of delivering to the chief clerk; see Figure 4)
  purpose           : ordering1
```

`ordering1`, in the role `purpose` in `being_obliged4`, is taken by substituting `ordering1` for |`query25`| in the `purpose` role in `being_obliged_function2` of Table 4.

Associated with every obligation are the conditions under which that obligation is fulfilled, and the conditions under which it is violated or not-violated. Obligation fulfillment conditions are dealt with in [1], [2]. Here we restrict our attention to violation and non-violation conditions.

In an obligation to pay for the consignment before 10 September 2002, the obligation is *violated* in the case that we count, on or after 10 September 2002, no (zero) payments for the consignment before 10 September 2002. That is, an occurrence of counting, on or after 10 September 2002, where zero payments for the consignment before 10 September 2002 are counted, brings about an occurrence of the obligation *being violated.* This is shown as `violating_function1` in Table 5.

3.2.2 Storing Privileges

Contractual provisions describe desirable, undesirable, and not-undesirable situations. The previous subsection showed how obligations are used to specify desirable occurrences, and how undersirable events, such as not paying by a deadline, may be captured by specifying what occurrences bring about violations. We now look at how not-undesirable events may be defined by specifying what types of events do *not* result in violations. The party that is immune from being in violation is said to have a *privilege*.

All obligations in law seem to confer the following implicit *privilege* (*immunity*): *the duty-bound party is immune from being in violation of the obligation as long as s/he still has opportunity to fulfill it*. Taking the obligation to pay by the 21^{st} of September, there is an implicit privilege to pay at any time before that date - that is, there being no deliveries at any stage before 21^{st} of September does not result in violation. More formally, there are zero occurrences of violating whose `source_occurrence` is a `counting` of zero (`0`) payments, where the counting occurs before 21^{st} September. This is shown in Table 6.

The next section describes how the semantics of a query may be stored in a database.

Table 6: Capturing the rule that not paying for the consignment before 21 September does not lead to violation

Occurrence	**Role**	**Participant**
`counting1`	`counted`	`query26`
	`count`	`0`

(`query26` = `occurrences of [violating] where [occurrences of [counting] where [<21 September] is [=occurred_on] intersection occurrences of [counting] where [=0] is [=count] intersection occurrences of [counting] where [query12] is [=counted]] is [=source_occurrence]`).

(`query12` is embedded in the obligation `being_obliged3`, and defined in Figure 2).

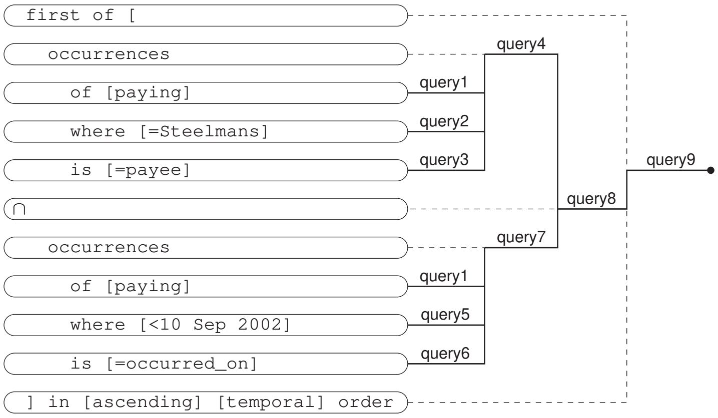

Figure 1: Parse tree for query that returns the first payment to Steelmans before 10 September 2002.

3.3 Storing Queries

To make queries that return occurrences more concise, we use our own language, EDEEQL[3]. Queries may be stored in occurrence-role-participant tabular form by assigning a query-identifier for each criterion's occurrence entry, and storing its type and value in the role and participant columns respectively. The criterion-value may be constant or a reference to an embedded query. The EDEEQL parser takes the textual form of the query and converts it to its tabular semantic form.

Take for example the query that returns the first payment for `consignment1`, before 10 September 2002, by SkyHi to Steelmans (`Query9` in the Participant column, for the row with `being_obliged1`, in Table 2). Figure 1 and Table 7 illustrate the parse tree for `query9`, and show its nested sub-queries. The amount of the payment, and the requirement that the payment be allocated to a particular debt [3], are omitted for simplicity in this depiction of the query.

The second query we need to store is "first occurrence of paying for `consignment1` before 21 September 2002" (`query12` in the `obliged` role for `being_obliged3` which was shown earlier). The complete parse tree for this query, excluding the repeated query sub-expressions from Figure 1 and Table 7, is given in Figure 2 and Table 8.

Finally, we also store the query `query17` (from `being_obliged2`) as shown in Figure 3 and Table 9, and we store the query `query22` (from `being_obliged4`) as shown in Figure 4 and Table 10. The nature of the item to be delivered is omitted for the sake of simplicity in the representations of these queries.

4 FINDING INCONSISTENT PROVISIONS

Deontic logic is the logic of actuality versus ideality, and deals with obligations and permissions. [2] contrasts our dynamic, occurrence-centric approach to the static view of Standard Deontic Logic. Notwithstanding these differences, previous work

Table 7: Storage schema for 'first payment to Steelmans before 10$^{\text{th}}$ September'

QueryID	CriterionType	Value
query1	type	paying
query2	identified-concept	Steelmans
query3	identified-concept	payee
query4	occurrence	query1
	participant	query2
	role	query3
query5	less-than	10 Sep 2002
query6	identified-concept	occurred_on
query7	occurrence	query1
	participant	query5
	role	query6
query8	intersectand	query4
	intersectand	query7
query9	set-criterion	query8

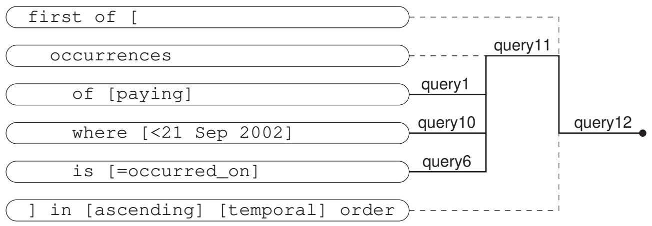

Figure 2: Parse tree for query that returns the first payment before 21 September 2002

in both deontic logic [13, 15, 16, 11] and the field of policy management [18, 14], points us to various possible types of conflicts between contractual provisions:

1. a set of occurrences may be both prohibited and permitted.
2. a set of occurrences may be both obliged and prohibited. Dynamic detection of such conflicts between obligations and prohibitions has been dealt with in [1].
3. an entity may be liable (in a Hohfeldian sense [10]) to fall into some legal state, but immune from falling into that state. For example, a party may be regarded as falling into violation, according to the contract, if they deliver 10 minutes late, but the *de minimus* rule of British law [19, p144] may, in contrast, see them as *not* falling into violation for such an immaterial deviation. Another example, which we shall deal with in more detail (Section 4.1), is the conflict between an obligation (that sees a violation existing if some occurrence doesn't happen by a deadline) and a privilege (which sees *no* violation happening in those circumstances).
4. a party may be empowered to bring about a legal state, but forbidden from doing so. A party is *empowered* if they are legally capable of bringing about a state of affairs. Being empowered does not necessarily imply being *permitted* to bring about that state of affairs [11].
5. a party may be obliged to bring about a legal state of affairs, but not empowered to do so.
6. a party may be obliged to bring about a state of affairs, but unable (e.g. through resource limitations) to bring about that state.

Table 8: Storage schema for 'first payment to Steelmans before 21^{st} September'

QueryID	**CriterionType**	**Value**
`query10`	`less-than`	`21 Sep 2002`
`query11`	`occurrence`	`query1`
	`participant`	`query10`
	`role`	`query6`
`query12`	`set-criterion`	`query11`

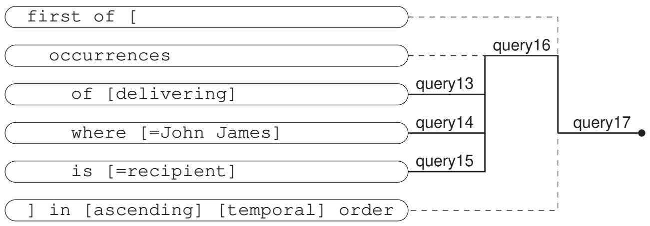

Figure 3: Parse tree for the query that returns the first occurrence of delivering to John James

There is also a further type of conflict not, to our knowledge, mentioned in the earlier literature:

7. obligations towards a single debt which contradict, or are unclear, as to what must be performed. The conflict is detected if the descriptions of the performances required to service an obligation *for a particular debt* are completely inconsistent or only partially overlapping.

Extending our previous work on conflict detection across business contracts and policies [1], we now illustrate, through worked examples, how we can detect inconsistencies of types 3 and 7.

4.1 Conflict between an obligation and a privilege (Type 3 conflict)

The fact that `query9` (first payment for `consignment1` before 10 September 2002) is a subset of `query12` (first payment for `consignment1` before 21 September 2002) is provable as follows [3]:

1. By Rule 3 `[< 10 September 2002]` (`query5`) is covered by query `[< 21 September 2001]` (`query10`)

2. By Rule 7 `[occurrences of [paying] where [< 10 September 2002] is [=occurred_on]]` (`query7`) is covered by `[occurrences of [paying] where [< 21 September 2002] is [=occurred_on]]` (`query11`)

3. By Rule 6 `query8` is covered by `query11`

4. By Rule 11 and step 3 it is evident that `query9` is covered by `query12`.

Now, a conflict becomes evident. Since the count of `query9` will always be zero when the count of `query12` is zero (because the former query is a subset of the latter), it appears, looking at `violation_function_1` of Table 5 and `counting1` of Table 6, that **Clause C.1** sees a violation in the case of no pre-September-10^{th} deliveries being counted after 10^{th} September, whereas **Clause P.3** sees no such violation in those circumstances, because Steelmans has the privilege that they may take until 21^{st} September to make the delivery. The conflict is illustrated diagrammatically in Figure 5.

[3] Each of the rules mentioned here is defined in detail in the Appendix.

Table 9: Storage schema for 'first delivery to John James'

QueryID	CriterionType	Value
query13	type	delivering
query14	identified-concept	John James
query15	identified-concept	recipient
query16	occurrence	query13
	participant	query14
	role	query15
query17	set-criterion	query16

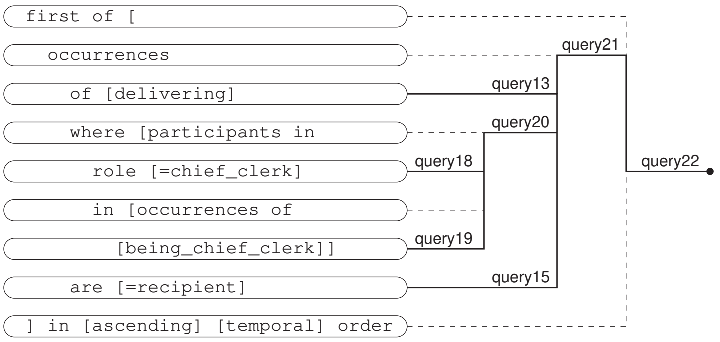

Figure 4: Parse tree for the query that returns the first occurrence of delivering to the chief clerk

4.2 Inconsistent descriptions of what is obliged (Type 7 conflict)

Inconsistencies often arise as to what is required to settle a debt. Here the descriptions of what must be done in exchange for something else overlap only partially or are completely non-overlapping. In our application scenario, the obligation to pay for `consignment1` by 10 September (`being_obliged1`) is inconsistent with the obligation to pay for `consignment1` by 21 September (`being_obliged2`) because it is unclear what must be done to settle the debt for the consignment. Notice, however, that the obligation to pay for `consignment1` by 10 September (`being_obliged1`) would *not* be inconsistent with the obligation to pay for another consignment, `consignment2`, by 21 September (call this say, `being_obliged4`), because the obligations are towards different debts and there is therefore no discrepancy as to what to do to satisfy *a particular debt*. To check for this type of conflict between obligations we therefore need to check a given obligation only against other obligations that are towards exactly the same debt.

Assume we have stored in our database both the obligation to pay for `consignment1` by 10 September (through the occurrence `being_obliged1`), and the obligation to pay for `consignment1` by 21 September (`being_obliged3`) as shown above. These obligations both pertain to the same consignment and are towards the same debt (`consignment1`). It can be seen above that `being_obliged1` nests `query9` and `being_obliged3` nests `query12`. Since, as shown in Section 4.1, these two queries (`query9` and `query12`) only partially overlap, there is a conflict between any obligations towards a particular debt that are associated with these queries, because the descriptions as to what to undertake to meet the obligation do not overlap exactly. It should be clear from this example that the conflict between the obligations is detectable in advance of any actual payments: even if there are currently no payments in the database, we can still determine analytically (as shown in Section 4.1) that there is only partial overlap between these two queries.

The conflict between the two statements of the obligation is most easily pictured visually, as show in Figure 6. Note again that both obligations shown in this diagram are for a single purpose or debt: both are directed at paying for `consignment1`. Clearly, a payment before 10^{th} September can fulfill both `being_obliged1` and `being_obliged3`, but a payment between 11^{th} and 21^{st} September violates `being_obliged1` yet fulfills `being_obliged3`.

Table 10: Storage schema for 'first delivery to the chief clerk'

QueryID	CriterionType	Value
query18	identified-concept	chief_clerk
query19	identified-concept	being_chief_clerk
query20	identified-concept	query18
	role	query19
query21	occurrence	query13
	participant	query20
	role	query15
query22	set-criterion	query21

Similarly, assume we had an obligation in the contract to pay for `consignment3` on 4th October and an obligation in the standard terms and conditions to pay for `consignment3` on 10th October. These obligations to pay for that debt contradict as it is unclear what to do to satisfy the obligations because the descriptions of the required performances do not overlap at all. This is illustrated in Figure 7.

Notice that, even though they may be towards the same purchased item, there is no conflict in installment payments because each payment is for a different installment (i.e. a different `purpose`). Consider the obligation to pay £5 for a consignment, `consignment4`, on 4th October and a further £10 for `consignment4` on 10th October. Here, the first obligation relates to `installment1` (on `consignment4`) and the second to `installment2` (on `consignment4`); even though the payments differ, the obligations do not conflict because they are for different purposes (`installment1` and `installment2` respectively). It may be that summing the payments or performing a net-present-value computation shows that the total paid for `consignment4` is in excess of that agreed, but identifying and resolving such over- or under-payment issues is a separate matter, may be dependent on organization-specific policy, and is not treated here.

5 TIME OF CONFLICT DETECTION

In our example of a conflict between contradictory obligations relating to payment for `consignment1`, the conflict was detected in advance of any actual payment, when it manifested itself at the time of delivery. The occurrence of delivering brought about a specific obligation (`being_obliged3`), according to the companies standard terms and conditions, which resulted in an *actual* conflict with an earlier contractual obligation (`being_obliged1`). Conflict detection at an earlier stage than performance should also be provided: we should be able to mechanically detect the *potential* conflict between the contract and the standard terms and conditions at the time the contract is being made, or at the time the standard terms and conditions are being altered, rather than merely at the time of delivery.

It is possible at contract- and policy-making time to determine what set of future occurrences would allow a contractual clause to be consistent with standard terms and conditions. Figure 8 illustrates how we can deduce what set of deliveries would result in the contractual obligation for `ordering1` being consistent with the obligation from the standard terms and conditions for `ordering1`. This calculation relies upon the application of the **Rule 10** (from the Appendix): remembering that − (minus) is the inverse operation of + (plus), we can prove that deliveries which occur on [21 September - 20 days] are the only deliveries which allow the obligation for `ordering1` from our contract to be consistent with our standard terms and conditions. Deliveries on any date other than that would make the obligations out of alignment with each other since the occurrences mandated by the obligations would no longer exactly overlap.

Figure 9 shows the various opportunities for conflict detection over the phases of contracting, for our example of the obligation to pay for delivery. *Potential* conflict may be detected at the time of contracting or policy-making; *actual* conflict between *prospective* views of different clauses describing the obligation may be detected at the time of delivery (the time of instantiation of specific, individual obligations); and *actual* conflict between *retrospective* views of the different clauses may be detected at the time of violation of one of the clauses.

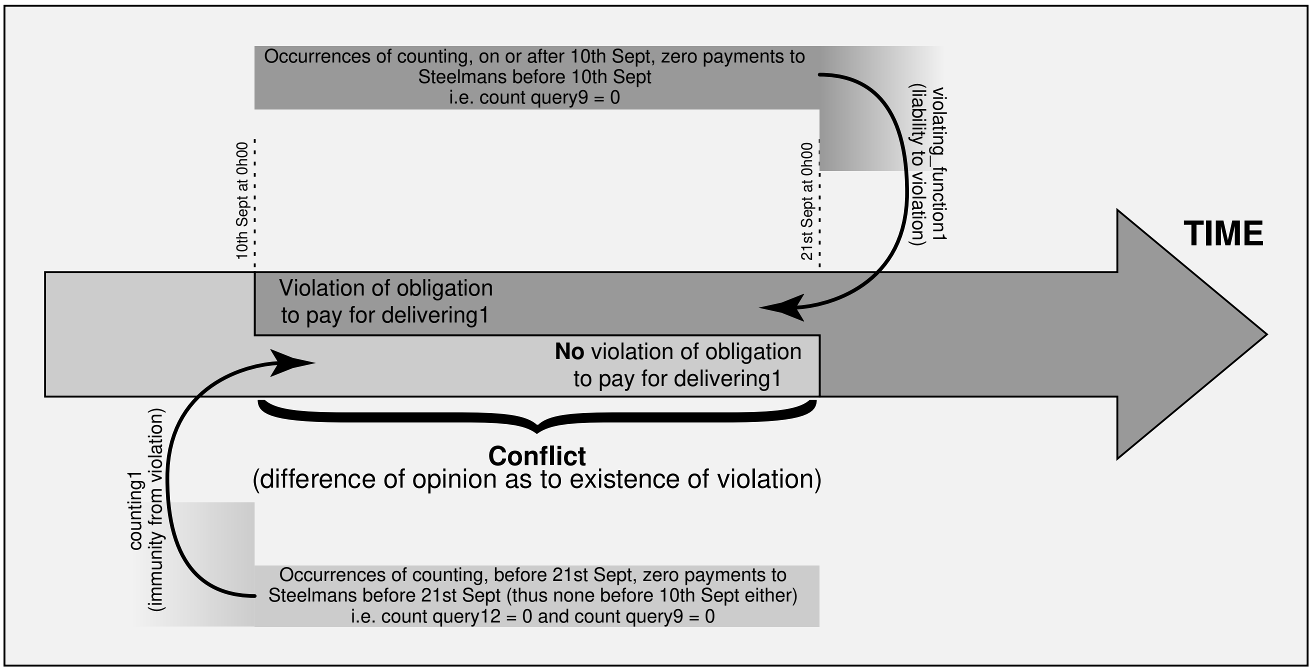

Figure 5: Conflict between an obligation (liability to violation) and an privilege (immunity from violation)

Table 11: Dirtied queries and their output dirt

Dirtied Query	Output Dirt
query19	being_chief_clerk1
query20	John James

(query19 = [occurrences of [being_chief_clerk]])
(query20 = participants in role [=chief_clerk] in [occurrences of [being_chief_clerk]])

Let us now look at an additional (simpler) example, which demonstrates how potential conflicts may be detected at the time of contracting.

First let us assume that John James is chief clerk (as depicted in Table 1). When being_chief_clerk1 is added to the database, we coverage-check it to determine which queries have their results altered by this data:

1. By Rule 1 being_chief_clerk1 is covered by [occurrences of [being_chief_clerk]] (query19)

2. By Rule 9 The query [occurrences of [being_chief_clerk]] (query19) dirties [participants in role [=chief_clerk] in occurrences of [being_chief_clerk]] (query20). Substituting the input dirt for the dirtied criteria (shown <u>underlined</u>) yields the partial re-evaluation query: [participants in role [=chief_clerk] in [<u>being_chief_clerk1</u>]]. Evaluation of this partial re-evaluation query yields the output dirt John James. query20 therefore covers John James and we can insert this fact in our cache of dirtied queries and their output dirt, as shown in Table 11.

Assume we have stored in our database both the obligation to deliver to John James (being_obliged2) and the obligation to deliver to the chief clerk (being_obliged4) as shown above. As shown earlier, the first obligation nests query17 (of Figure 3), and the latter nests query22 (of Figure 4). Both obligations are in respect of ordering1 (that

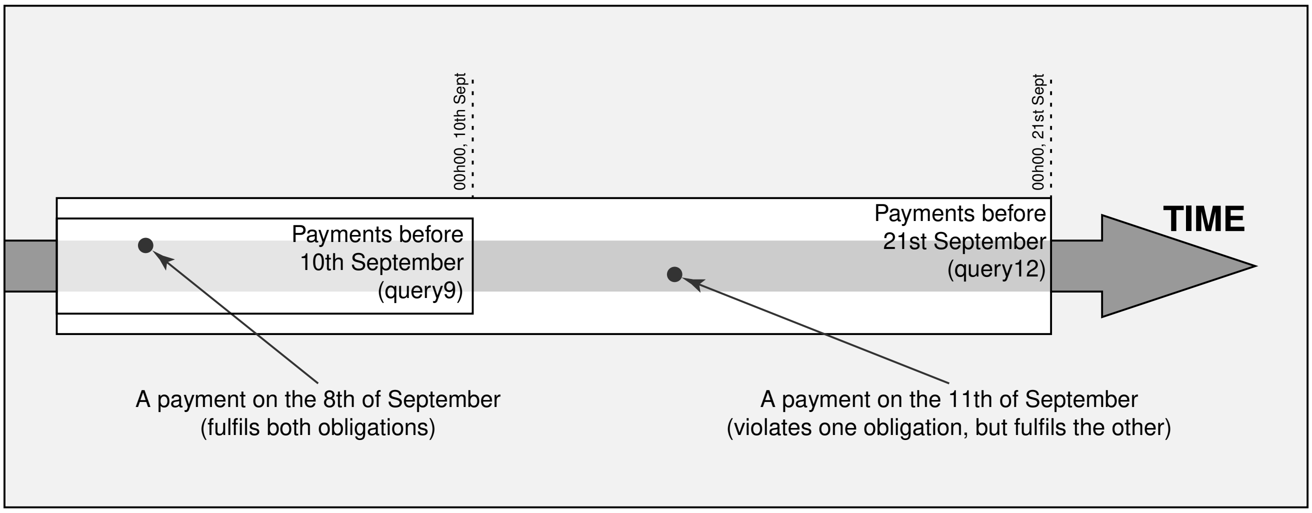

Figure 6: Inconsistent obligations: Different descriptions of occurrences that can satisfy debt for `consignment1`

is, have `ordering1` as their `purpose`). Assuming John James is the chief clerk, it can be shown that `query22` partially overlaps `query17`:

1. By Rule 5 `[=John James]` (`query14`) is covered by any query covering John James. `John James` is covered by `query20` (`participants in role [=chief_clerk] in occurrences of [being_chief_clerk]`), as shown in the last row of the 'dirtied queries and output dirt' cache shown in Table 11. Therefore `query20` covers `query14`.

2. By Rule 7 and step 1 we see the query `occurrences of [delivering] where [participants in role [=chief_clerk] in occurrences of [being_chief_clerk]] are [=recipient]` (`query21`) covers `occurrences of [delivering] where [=John James] is [=recipient]` (`query16`).

3. By Rule 11 and steps 1 and 2 `query22` partially overlaps `query17`

As the obligations `being_obliged2` and `being_obliged4` have the same `purpose`, but refer to potentially different performances – because `query17` and `query22` overlap only partially – there is potential for conflict. The potential conflict can be visualized as shown in Figure 10: the set of deliveries to anyone who is ever a chief clerk includes the set of deliveries to John James (who is currently chief clerk) and the set of deliveries to Mary Moses (who may in the future be chief clerk). Consider the case where John James is promoted to operations manager and Mary Moses becomes chief clerk. In this circumstance the contract mandates delivery to John James, but the standard terms and conditions mandate delivery to Mary Moses.

It should be noticed that here we have detected the *potential* conflict at the time of contracting. Notice that the potential conflict only becomes an actual conflict when John James ceases to be chief clerk, since at that time 'deliveries to John James' becomes disjoint from 'deliveries to the chief clerk'. Assuming we recorded in the database a new fact, that John James is now disjoint from the set of clerks: i.e. that `query14` is now disjoint from `query20`, then the conflict becomes *actual and unavoidable* as deliveries to John James is now disjoint from deliveries to the chief clerk, as shown in Figure 11.

The example of obligations to deliver, given above, reinforces the organizational design principal that, unless the individual possesses unique properties and is personally required, reference should be made to roles, rather than to particular individuals who hold those roles. Specifications that mention roles are more resilient to changes in role-players as individuals vacate their positions in organizations [6]. Similarly, referring to an item by its attributes, instead of by its identity, may improve stability as new instruments, machines or technologies replace older ones.

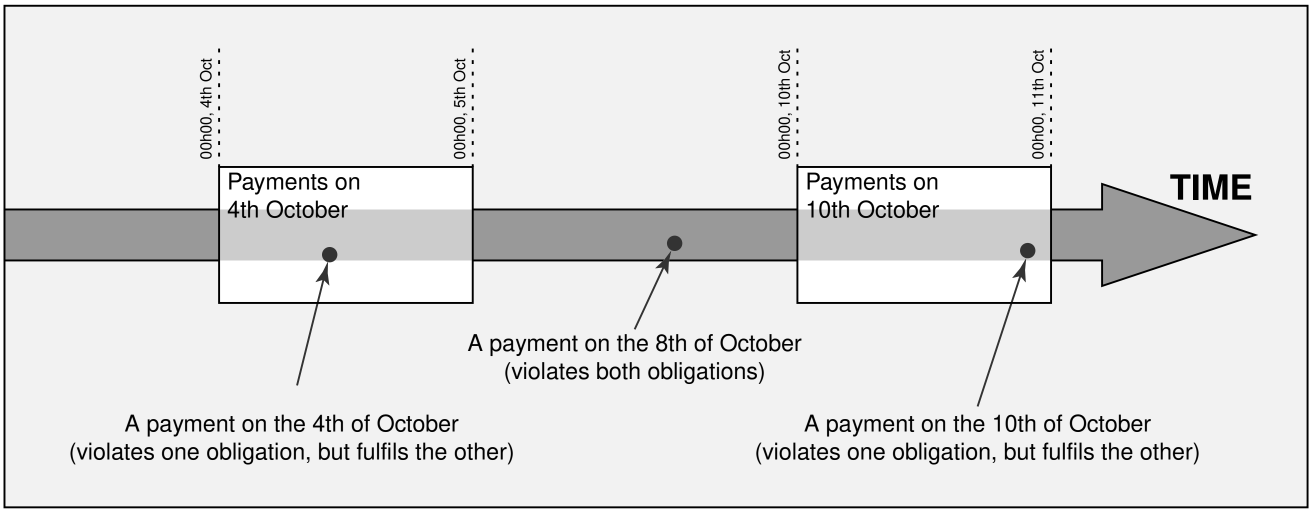

Figure 7: Inconsistent obligations: Different descriptions of occurrences that can satisfy debt for `consignment3`

6 CONCLUSION

We have proposed a coverage-determination mechanism for queries within e-service environments. We discussed the data and query storage techniques employed by the EDEE system, and through worked examples, demonstrated how our approach efficiently determined conflicts which appeared dynamically between obligations and privileges, and between obligations towards a given debt, defined across business contracts and organizational policies.

ACKNOWLEDGMENTS

This research is supported by grants from the Cambridge Commonwealth and Cambridge Australia Trusts, the Overseas Research Students Scheme (UK), and the University of Cape Town Postgraduate Scholarships Office. We are grateful to Microsoft Research Cambridge for funding a continuation of the research.

Thanks are due to Dr Ken Moody for helpful comments on the draft.

References

[1] A. Abrahams, D. Eyers, and J.M. Bacon. A coverage-determination mechanism for checking business contracts against organizational policies. In *Proceedings of the Third VLDB Workshop on Technologies for E-Services (TES'02)*, 2002.

[2] A.S. Abrahams and J.M. Bacon. The life and times of identified, situated, and conflicting norms. In *Sixth International Workshop on Deontic Logic in Computer Science (DEON'02), Imperial College, London, UK*, May 2002.

[3] A.S. Abrahams and J.M. Bacon. A software implementation of Kimbrough's disquotation theory for representing and enforcing electronic commerce contracts. *Group Decision and Negotiations Journal*, Forthcoming.

[4] R. Ayres and P. J. H. King. Querying graph databases using a functional language extended with second order facilities. In *Advances in Databases, Proceedings of the 14th British National Conference on Databases, (BNCOD 14)*, pages 189–203, Edinburgh, UK, July 1996. Springer.

[5] R.W.H. Bons, R.M. Lee, R.W. Wagenaar, and C.D. Wrigley. Modelling inter-organizational trade procedures using documentary petri nets. In *Proceedings of the Hawaii International Conference on System Sciences*, 1995.

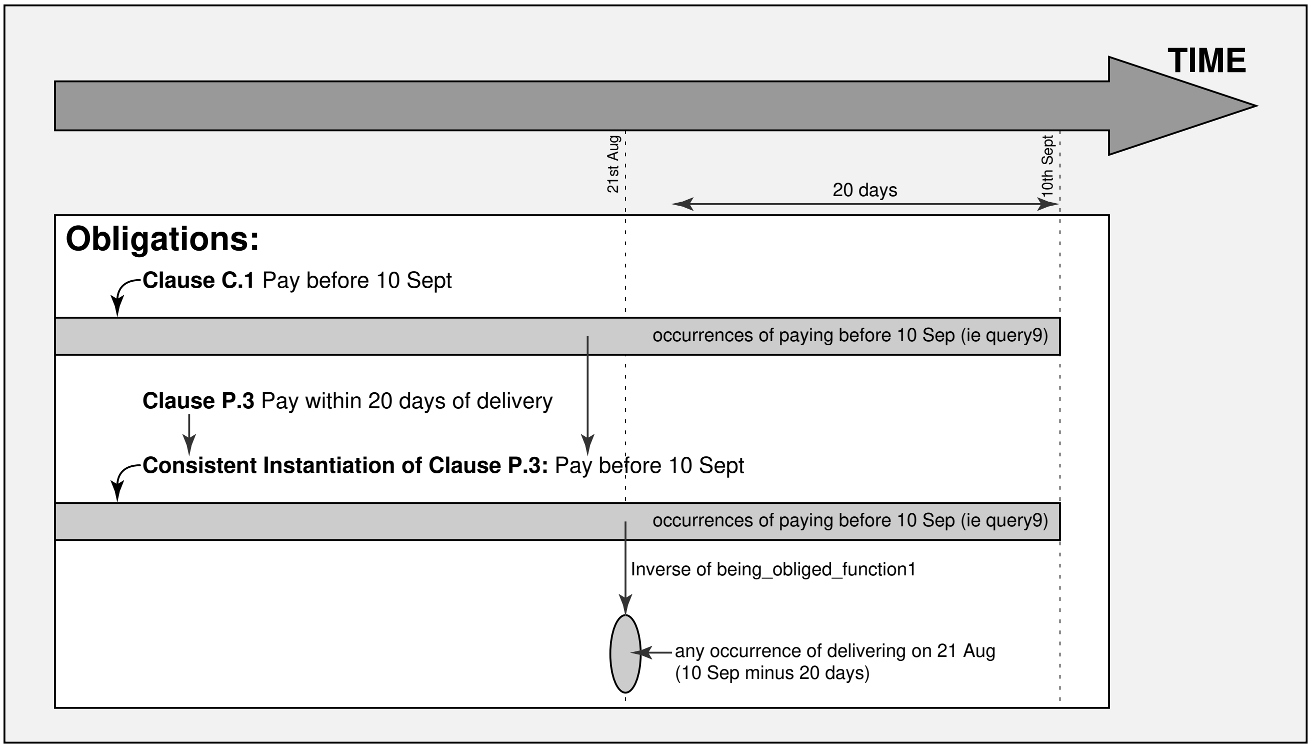

Figure 8: Deducing which operational occurrences would allow consistent obligations for the obligations pertaining to `ordering1`

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–38, 2001.

[7] A. Daskalopulu, T. Dimtrakos, and T.S.E. Maibaum. E-contract fulfillment and agents' attitudes. In *Proceedings ERCIM WG E-Commerce Workshop on the Role of Trust in E-Business*, Zurich, October 2001.

[8] C. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[9] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering Bulletin*, 18(2):3–18, 1995.

[10] W.N. Hohfeld. *Fundamental Legal Conceptions as Applied in Judicial Reasoning*. Greenwood Press Publishers, 1978.

[11] A.J.I. Jones and M. Sergot. A formal characterisation of institutionalised power. *Journal of the Interest Group in Pure and Applied Logic*, 4(3):427–443, 1996.

[12] S.O. Kimbrough. Reasoning about the objects of attitudes and operators: Towards a disquotation theory for the representation of propositional content. In *Eight International Conference on Artificial Intelligence and the Law (ICAIL 2001)*, St Louis, Missouri, May 2001.

[13] R.M. Lee. Bureaucracies as deontic systems. *ACM Transactions on Office Information Systems*, 6(2):87–108, April 1988. Special Issue on the Language/Action Perspective.

[14] E.C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, November/December 1999. Special Section: Managing Inconsistency in Software Development.

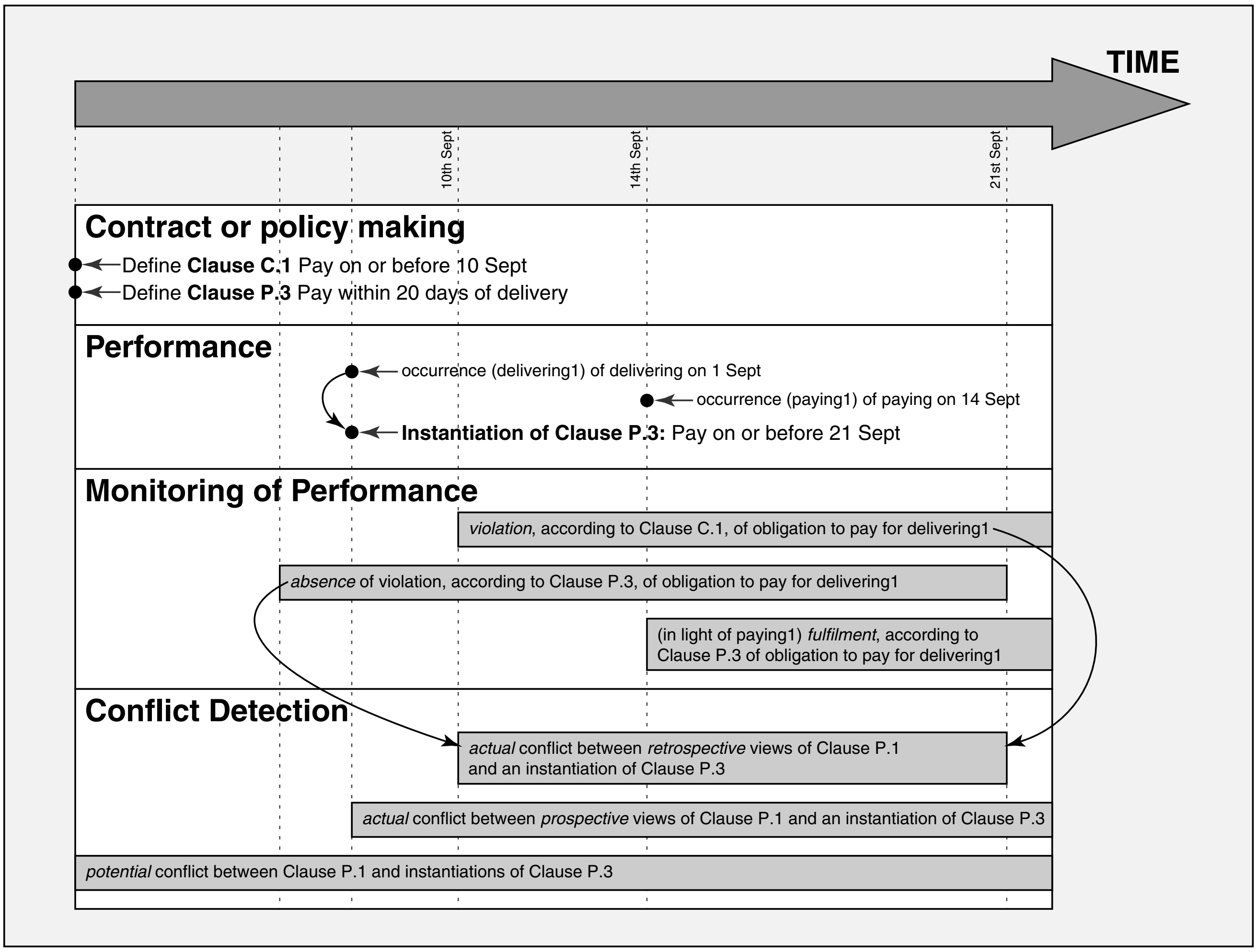

Figure 9: Time of conflict detection over the phases of contracting

[15] D. Makinson. On the formal representation of rights relations. *Journal of Philosophical Logic*, 15:403–425, 1986.

[16] D. Makinson. Rights of peoples: Point of view of a logician. *The Rights of Peoples*, 1988.

[17] D. P. Miranker. TREAT: A better match algorithm for AI production system matching. In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 42–47, Seattle, WA, July 1987. Morgan Kaufmann.

[18] J.D. Moffett and M.S. Sloman. Policy conflict analysis in distributed system management. *Journal of Organizational Computing*, April 1993.

[19] C.P. Thorpe and J.C.L. Bailey. *Commercial Contracts*. Kogan Page Limited, 1999.

A COVERAGE CHECKING RULES

Below are the rules used for determining coverage relationships between queries in our examples (the complete list is available on request).

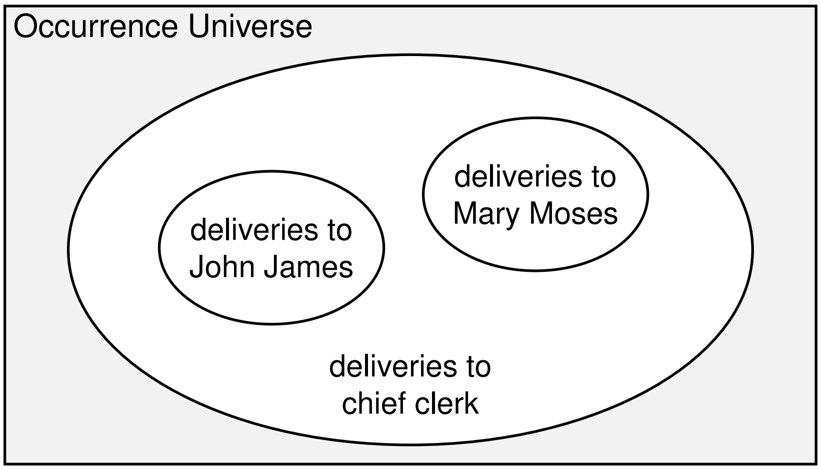

Figure 10: Potentially inconsistent obligations: Different descriptions of occurrences that are obliged under `ordering1`

Rule 1 An item is covered by queries with matching `type` criteria.

Rule 2 Transitively, a query is covered by any coverer of its coverers.

Rule 3 A temporal equal-to query Q, is covered by an equal-to, less-than or greater-than query P if P's `equal-to`, `less-than`, or `greater-than` criterion is, respectively, equal to, greater than, or less than Q's `equal-to` criterion. A temporal 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 temporal greater-than query Q, is covered by temporal greater-than queries where the `greater-than` criterion is less than the `greater-than` criterion of Q.

Rule 4 A participant, occurrence, or role is covered by concept-identification queries where the `identified-concept` criterion is identical to the participant, occurrence, or role identifier.

Rule 5 A concept-identification query is covered by any query that covers its `identified-concept` criterion.

Rule 6 An intersection query Q, is covered by any intersection query P, if each of P's `intersectands` covers some non-zero number of Q's `intersectands`. Similarly, an intersection query Q is covered by a query P, if P covers some non-zero number of Q's `intersectands`.

Rule 7 For two participant queries[4], P covers Q if P's `role` criterion covers Q's and P's `occurrence` criterion covers Q's. Similarly for occurrence queries[5].

Rule 8 An ordinal (sequence) query is covered by its `set` criterion. e.g. `first [occurrences of [paying]]` is covered by `occurrences of [paying]`.

Rule 9 A query, Q, dirties any participant or occurrence query that has Q as its `participant` criterion, `occurrence` criterion, or `role` criterion.

Rule 10 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

[4] EDEEQL syntax is: `participant_query = PARTICIPANTS IN ROLE role_criterion IN occurrence_criterion`

[5] EDEEQL syntax is: `occurrence_query = OCCURRENCES OF occurrence_criterion WHERE participant_criterion IS | ARE role_criterion`

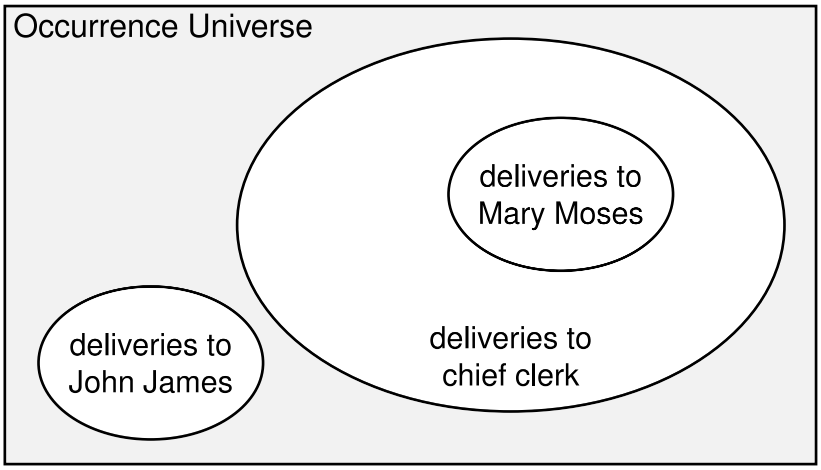

Figure 11: Actual conflict between an obligation to deliver an order to John James (who is not chief clerk anymore) and an obligation to deliver the order to the chief clerk

letter. This has the inverse operation `firstLetterOf()` which maps a string to its first letter[6]. Assume the item, y, which 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 `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')`. If the operation function, Ω, is injective – i.e. one-to-one: no two different inputs give the same output – then we can conclude that, for any parameter other than the computed parameter (i.e. for any parameter other than '`B`' in this example), the description is *disjoint* from the item y. e.g. We can conclude that `symbolsStartingWith(<>B)` is disjoint from '`Brian`'. This method of using inverse operations to find descriptions (operations and the specific parameter values) that cover or are disjoint from an item can be generally applied. Furthermore, as far as we are aware, this is be a novel mechanism of finding descriptions for items.

Rule 11 Provided the ordering criteria are identical for ordinal queries P and Q, and both P and Q select the first x elements by such criteria, P covers Q if P's `set` criterion covers Q's `set` criterion.

[6]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 a 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.