A Coverage-Determination Mechanism for Checking Business Contracts Against Organizational Policies

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

Computer Laboratory
University of Cambridge
{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 occurrences promised and prohibited under the provisions of the contracts and policies of an organization. This paper proposes a mechanism for both the static and dynamic derivation of the overlaps between queries. We show, through worked examples, that by determining these covering relationships we can discover inconsistencies between business contracts and organizational policies.

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 [2] 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 a promise, prohibition, permission or power; only the former two are demonstrated in this paper. Each provision embeds a query which describes the promised or prohibited occurrences. 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 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 [9] 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.

This paper shows how conflicts may be detected at the time contracts are added to the database, or when inserted data dirties queries and thus brings provisions into conflict. The example we present shows that a potential conflict between a promise to pay and a prohibition against a particular type of payment can be flagged as soon as the *promise* is entered, rather than merely at the time of payment. This conflict might be resolved by breaking the promise, violating the prohibition, or voiding one or both. Such conflict resolution is treated in [1].

2 Related Work

Current contract-driven inter-enterprise workflow architectures, such as COSMOS [12] and CrossFlow [11] focus on service advertisement and invocation, but do not ascertain consistency between contractual terms and business policies. Initiatives such as the OASIS ebXML Collaboration Protocol Profile (CPP) and Agreement (CPA) specifications [7] again provide service advertisement and conformance-checking framework specification for the negotiation of organizational inter-operation. CPAs capture the technology-specific 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 any provision for fulfillment monitoring. CPAs define specific business process arrangements, rather than a framework for managing the potentially conflicting policy sources which may govern a single business entity.

Previous contract assessment approaches, such as [4, 6], 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. The goal of the OASIS Provisioning Technical Committee [5] 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) 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.

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 [2] 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 [3]), 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 [13] 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 matching in RETE and TREAT to also perform pattern/pattern matches as well.

3 Application Scenario

We introduce an application scenario, describe how operational data, provisions and queries are stored, then illustrate via a worked example how conflicts between provisions and internal regulations are determined.

In our scenario, SkyHi Builders is a construction company. Steelmans Warehouse a supplier of high-grade steel. SkyHi, having recently won a tender to build a new office block, enters into a contract with Steelmans. We select a hypothetical clause from this contract, **Clause C.1** SkyHi promises to pay Steelmans £25,000, and a clause from the SkyHi's risk management procedures (i.e. internal policy), **Clause P.3** Payments of more than £10,000 to suppliers are prohibited.

3.1 Storing Operational Data

Let us say SkyHi, a customer of Steelmans, has paid Steelmans £25,000 for a specific shipment. Let `being_supplier1` and `paying1` denote instances (hence the `1` added to create a unique identifier) of occurrences of type *being a supplier* and *paying* respectively. Table 1 shows the occurrence, role, participant schema

Table 1. A tabular schema for storing various occurrences

Commentary	Occurrence	Role	Participant
Steelmans *being a supplier* for SkyHi	being_supplier1	supplier	Steelmans
		supplied	SkyHi
SkyHi *paid* £25,000 to Steelmans	paying1	payer	SkyHi
		paid_amount	£25,000
		payee	Steelmans

Table 2. Schemas for storing prohibitions and promises

Occurrence	Role	Participant
prohibiting1	prohibited	Query10
promising1	promised	Query19

(Query10 = occurrences of paying with over £10,000 in role paid ∩ occurrences of paying with a supplier in role payee. See figure 1),
(Query19 = first occurrence of SkyHi paying Steelmans £25,000. See figure 2)

employed in Edee to store this operational data. 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.

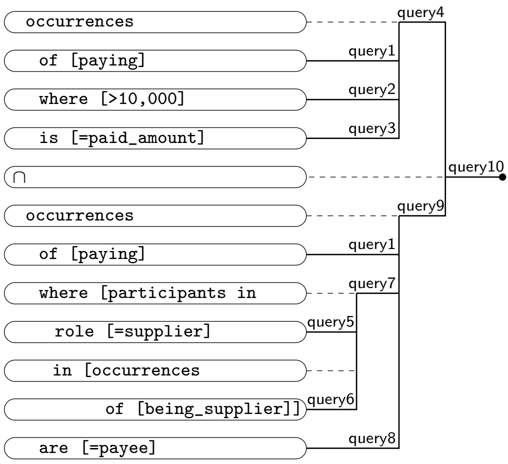
3.2 Storing Provisions of Contracts and Policies

To store contractual provisions – e.g. "X *prohibits* that [Y be paid]" and "X *promises* that [X pay Y]" – in a relational database we need to handle their embedded propositional content [2, 10].

Consider Clause P.3 from the application scenario presented above. Clearly, we cannot store simply "Steelmans prohibits [paying1]" because paying1 is a concrete instance and might not yet have occurred anyway. We instead store the prohibition as `prohibiting1` in Table 2, and indicate the prohibited occurrences using a pointer to a database view (query) describing the set of prohibited occurrences, which is `query10` in Figure 1. Note that this query would be empty in the case that no prohibited occurrences exist.

Similarly, the promise in Clause C.1 of our scenario cannot be stored via "SkyHi promises paying1", because we need to store a description of a payment. The promise is thus stored as `promising1` in Table 2 with the promised occurrence represented by the pointer to `query19` in Figure 2. `Query19` asks for the *first* payment since it is exactly one payment that is promised. It may be empty in cases where the promise is broken or voided and no payments are made.

Storing provisions therefore requires the storage of views or queries which describe the promised or prohibited occurrences. Conflicts can be detected from



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

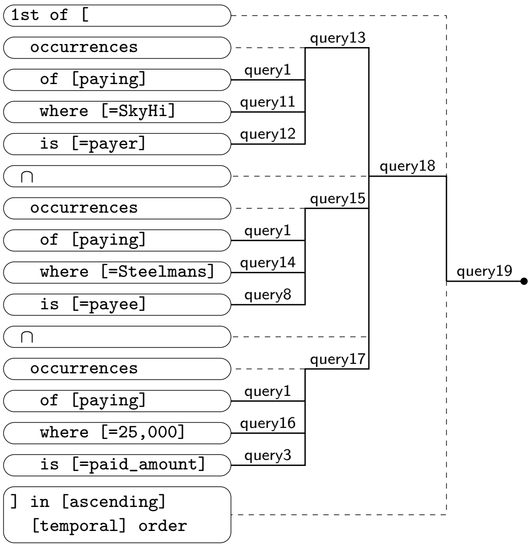
Fig. 1. Parse tree and storage schema for query that returns all occurrences where more than £10,000 is paid to a supplier

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

3.3 Storing Queries

To make queries that return occurrences more concise, we use our own language, EdeeQL[2]. 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 all occurrences where more than £10,000 is paid to a supplier (`query10`, in the `Participant` column, for the row with `prohibiting1`, in Table 2 above). Figure 1 illustrates the parse tree for `query10`, and shows its nested sub-queries (Currency representation is omitted for simplicity). The second query we need to store is "select the first payment of £25,000 by SkyHi to Steelmans" (`Query19` in the `Participant` column, for the row with `promising1`, in Table 2 above). The complete parse tree for this query, excluding the repeated query sub-expressions shown earlier, is given in Figure 2. Storing queries explicitly is helpful for finding covering-queries since we can analytically determine which queries, among a large number of stored queries, cover a certain item or query. We describe the mechanism for finding covering-queries, in the next section.



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

Fig. 2. Parse tree and storage schema for query that returns the first payment of £25,000 by SkyHi to Steelmans

4 Finding Overlapping and Inconsistent Provisions

Assume a prohibition, `prohibiting1`, and the associated query describing the prohibited occurrences are stored in an empty database as shown in Table 2 and Figure 1. We then record that Steelmans is a supplier of SkyHi by inserting the occurrence, `being_supplier1`, as described in Table 1 (assume the payment, `paying1`, is not ever inserted). Upon insertion, the coverage-checking algorithm examines each of the unique items `Steelmans`, `being_supplier1`, `supplier`, `SkyHi`, and `supplied` in the set of triples added for this occurrence[1]:

1. **By Rule 4** item `supplier` is covered by the query `[=supplier]`.
2. **By Rule 1** item `being_supplier1` is covered by the query `occurrences of [being_supplier]`.
3. **By Rule 9** queries `[=supplier]` and `occurrences of [being_supplier]` dirty the query `[participants in role [=supplier] in occurrences of [being_supplier]]`. Substitution of the input dirt for the dirtied criteria (shown underlined) 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 9 and step 3** Item (`Steelmans`) dirties query `occurrences of paying where [participants in role [=supplier] in occurrences of [being_supplier]] are [=payee]`. Substitution of the input dirt (shown

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

Table 3. Dirtied queries and their output dirt after addition of occurrence of `being_supplier1` to a new data-store

Dirtied Query	Output Dirt
query5	supplier
query6	being_supplier1, . . .
query7	Steelmans

(query5 = [=supplier]), (query6 = occurrences of [being_supplier]), (query7 = [participants in role [=supplier] in occurrences of [being_supplier]])

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.

We conclude that the new occurrence, `being_supplier1`, is not prohibited, since the only query that covers it is, `occurrences of [being_supplier]`, which is not in the `prohibited` role in any prohibition. We nevertheless record which queries were dirtied by this new data and cache the output dirt, since we can use this dirt in future partial re-evaluations. The dirtied queries and their output dirt is shown in Table 3. The incremental nature of the algorithm is important as tens of thousands of occurrences may be stored in the database. Re-executing every stored query on each occurrence addition is infeasible, particularly since most results will be unchanged. The cache of dirtied queries facilitates 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 only re-evaluate 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.

Say SkyHi promises to pay to Steelmans £25,000. Assume this payment has been contemplated, but not effected; no occurrence of `paying` has been added to the data store. As shown in Table 2 and Figure 1, the promise can be represented by embedding the stored query, `query19`, in an occurrence of *promising*. Now, comparing the description of the promised occurrences (`query19`) to other stored queries, proceeding from its most deeply nested sub-expressions upwards:

1. **By Rule 3** `[=25,000]` (`query16`) is covered by query `[>10,000]` (`query2`)
2. **By Rule 7** `occurrences of paying where [>10,000] is [=paid_amount]` (`query4`) covers the query `occurrences of paying where [=25,000] is [=paid_amount]` (`query17`)
3. **By Rule 5** `[=Steelmans]` (`query14`) is covered by any query which covers `Steelmans`. As seen earlier, `[participants in 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 3. Therefore `query7` covers `query14`.

4. By Rule 7, and step 3 `occurrences of paying where [participants in role [=supplier] in occurrences of [being_supplier]] are [=payee]` (`query9`) covers `occurrences of paying where [=Steelmans] is [=payee]` (`query15`)

5. By Rule 6, step 2 and step 4 `Query18` is covered by `Query10`

6. By Rule 8 and step 5 The `set` criterion (`Query18`) covers `Query19`

7. By Rule 2, step 5 and step 6 `Query19` is covered by `Query10`.

We have thus shown that what is promised (the description of the promised occurrences = `Query19`) in this context is covered by what is prohibited (the description of the prohibited occurrences = `Query10`). We have thus detected a dynamically appearing *conflict* between a provision embedded in a contract, and an organizational policy. [1] describes mechanisms for resolving such conflicts. Research into increasing the efficiency of our implementation, and confirming through performance tests that the system is comfortably suited to real-world business workloads, is ongoing.

5 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 a worked example, demonstrated how our approach efficiently determined conflicts which appeared dynamically between business contracts and organizational policies.

6 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.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.

[2] 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.

[3] R. Ayres and P. J. H. King. Querying graph databases using a functional language extended with second order facilities. In *Advances in Databases, 14th British National Conferenc on Databases, BNCOD 14, Edinburgh, UK, July 3-5, 1996, Proceedings*, pages 189–203. Springer, 1996.

[4] 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 Internaional Conference on System Sciences*, 1995.

[5] OASIS Provisioning Services Technical Committee. An introduction to the provisioning services technical committee. `http://www.oasis-open.org/committees/provision/Intro-102301.doc`, 2001.

[6] 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.

[7] OASIS ebXML Collaboration Protocol Profile and Agreement Technical Committee. Collaboration-protocol profile and agreement specification. `http://www.oasis-open.org/committees/ebxml-cppa/documents/ebcpp-2_0.pdf`, 2002.

[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] 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.

[11] M. Koetsier, P. Grefen, and J. Vonk. Cross-organisational workflow: Crossflow ESPRIT E/28635 contract model, deliverable D4b. Technical report, CrossFlow consortium, 1999.

[12] M. Merz, E. Griffel, T. Tu, S. Muller-Wilken, H. Weinreich, M. Boger, and W. Lamersdorf. Supporting electronic commerce transactions with contracting services. *International Journal of Cooperative Information Systems*, 7(4):249–274, December 1998.

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

A Coverage Checking Rules

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

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 numeric equal-to query Q, is covered by an equal-to, less-than or greater-than query if its `equal-to`, `less-than`, or `greater-than` criterion is, respectively, equal to, greater than, or less than Q's `equal-to` criterion. 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.

Rule 4 Any 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`.

Rule 7 For two participant queries[2], P covers Q if P's `role` criterion covers Q's and P's `occurrence` criterion covers Q's. Similarly for occurrence queries [3].

Rule 8 An ordinal (sequence) query is covered by its `set` criterion. e.g. `1st [payments]` is covered by `payments`.

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

[2] EdeeQL syntax is: `participant_query = PARTICIPANTS IN ROLE role_criterion IN occurrence_criterion`

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