Querying Contract Databases Based on Temporal Behavior

Elio Damaggio
UC San Diego
elio@cs.ucsd.edu

Alin Deutsch
UC San Diego
deutsch@cs.ucsd.edu

Dayou Zhou
UC San Diego
dzhou@cs.ucsd.edu

ABSTRACT

Considering a broad definition for service contracts (beyond web services and software, e.g. airline tickets and insurance policies), we tackle the challenges of building a high performance broker in which contracts are both specified and queried through their temporal behavior. The temporal dimension, in conjunction with traditional relational attributes, enables our system to better address difficulties arising from the great deal of information regarding the temporal interaction of the various events cited in contracts (e.g. "No refunds are allowed *after* a reschedule of the flight, which can be requested only *before* any flight leg has been used"). On the other hand, querying large repositories of temporal specifications poses an interesting indexing challenge. In this paper, we introduce two distinct and complementary indexing techniques that enable our system to scale the evaluation of a novel and theoretically sound notion of *permission* of a temporal query by a service contract. Our notion of permission is inspired by previous work on model checking but, given the specific characteristic of our problem, does not reduce to it. We evaluate experimentally our implementation, showing that it scales well with both the number and the complexity of the contracts.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*Data Model*; J.1 [**Computer Applications**]: Administraitive Data Processing—*Business, Law*

General Terms

Algorithms, Perfomance, Theory

Keywords

temporal behavior, contracts, Linear Temporal Logic, LTL, indexing, Büchi automata

1. INTRODUCTION

Service contracts are seldom completely represented by a set of predefined attributes and as such are usually modeled only partially in IT systems. As an example, consider the market of airfares. Every plane ticket is actually a complex contract with dozens of conditions regarding validity, refundability and changeability, among others. Some fixed categories exist (e.g. economy tickets, business, first class), but they do not express all the subtleties of the contracts that the customers are required to sign. Any non-standard travel arrangement requires sifting through the full text of the contracts of the fares returned by such brokers as Expedia, Orbitz and Travelocity, or asking an expert travel agent. Generally, analogous characteristics are found in service markets where there is no negotiation of the contracts, but that present many possible choices in direct competition (e.g. airfares, insurances, warranties).

SIGMOD'11, June 12–16, 2011, Athens, Greece.

EXAMPLE 1. *Over 100 airlines operate in the US market [26], each of them offering around 20 different fares [27], [24]. Fare rules specify a great deal of temporal conditions such as (from the United Airlines Fare Contract [1]): '1) Passenger may change the routing and/or the ultimate destination [...] provided that, after transportation has commenced, a one-way ticket will not be converted into a round-trip, circle-trip, or openjaw trip ticket.' Note, also, that voluntary rerouting might be possible only by reissuing a ticket with a different 'booking code' which in turn has to abide to additional restrictions such as minimum stays and advanced purchase requirements [24].*

In this work, we envision a contract brokering system that, in addition to standard relational attributes, allows contracts to be specified and queried by their temporal behavior. For illustration purposes, we introduce now a running example in the airfares scenario. Airlines, as contract providers, register all their fares on our brokering system. Each airfare specifies its temporal aspects, in addition to the usual relational attributes (e.g. baggage conditions, prices). These aspects involve how tickets are changed (if and when it is possible), how the customers can be refunded and so on. A possible query could look for: *the cheapest fare on 10/19/2010 from San Diego to New York, that allows a partial ticket refund or a date change after the first leg has been missed.* In order to answer this query, our broker would first retrieve the list of fares available from San Diego to New York, along with their prices, using a standard DBMS. It would then use the temporal specifications to filter the fares that do not satisfy the temporal portion of the query (e.g. '... that allows a date change even in the case of a missed flight'). In this work, we focus on the temporal aspect of our system.

EXAMPLE 2. *Assume that all the airfares* 'from San Diego to New York on 10/19/2010' *returned by the DBMS fall into three categories of tickets (e.g. United Business, AA Economy for Platinum Clients) that differ in their policies regarding refunds and changeability. Their contract restrictions are the following:*

Ticket A
1. *No refunds are allowed after date changes*
2. *Unlimited date changes*

Ticket B
1. *Refunds always allowed*
2. *Date changes only before the scheduled departure*

Ticket C
1. *No refunds are allowed*
2. *Only one date change is allowed*
3. *Date changes only before the scheduled departure*

Our system has to identify which of these contracts satisfy the temporal portion of the query: 'allows a partial ticket refund or a date change after the first leg has been missed'. *Intuitively, the system should return* ***Ticket A*** *(because it would allow a reschedule) and* ***Ticket B*** *(that would allow a refund). It should not return* ***Ticket C*** *because it does not allow any refund nor rescheduling after a missed flight.*

Our design is guided by the following requirements:
i) Specifications and queries need to be expressive enough to capture realistic temporal behavior,
ii) The interface between customers and providers should be compact and reasonably stable,
iii) Published contract specifications should not require revisions if a contract with a different policy is published,
iv) Our representation for contracts and queries should declare a set of declarative clauses, in order to loosely follow natural language specifications.

Modeling complex temporal behavior with relational attributes is tempting but falls short in all stated requirements for the following reasons. Let us consider a customer's specific requirement regarding the refundability options of a partially used ticket (as in Example 2). It would be easy to add an attribute 'refundable' with a set of predefined values that code the various situations: 'no refunds', 'only full refunds', 'partial refunds allowed'. The problem lies in the interaction of these categories with other ticket features like changeability, e.g. a refund might not be available for a ticket that was changed. Moreover, both aspects interact with the various phases of the trip (e.g. before or after the first flight leg was completed or missed). In principle, every new aspect can interact with all the others, leading to an exponential blow up of the number of possible interaction 'categories' (e.g. 'partial refunds allowed only if requested before the date of the first flight leg', 'partial refunds allowed even if the first leg is missed'). These categories then become too numerous to specify as schema attributes. They are hard to define and specify for the contract providers and, even more importantly, hard for the consumer to understand, leading either to a lack of expressiveness (violating requirement *i*) or a bloated interface (*ii*). Moreover, publishing contracts with policies that differ from the existing ones might change the relational schema, adding and/or modifying attributes. This would force the revision of previously published contracts (*iii*). Lastly, the relational schema resulting from such a modeling would be very far from the original informal description (*iv*).

Our proposal considers a common vocabulary of events (e.g. 'ticket refunded', 'flight used', 'flight missed', 'date changed') that is small enough to overview and will act as the interface between providers of contracts and customers that query them. Analogously to the real world, the temporal behavior of contracts is specified as a set of declarative clauses referring to the common vocabulary, as are the temporal properties required by the customer. The vocabulary refers to a set of familiar events pertaining to the application domain, and is much more compact than the rich family of temporal behaviors usually expressed as declarative clauses in contracts (e.g. 'allowing a partial refund after the first leg is used'). Intuitively, a contract allows only certain temporal *sequences* of events. The allowed sequences have to simultaneously satisfy all the clauses in the contract. When interested in a particular property of a contract (e.g. 'allowing a partial refund after the first leg is used'), customers formulate an ad-hoc query against the vocabulary, detailing the property of a desired temporal sequence of events. The query simply returns all contracts that allow a sequence satisfying the desired property. We say that such contracts *permit* the query.

EXAMPLE 3. *We use the following event vocabulary to crudely model single-trip flights:* purchase *(ticket purchased),* use *(ticket used),* missedFlight *(customer missed the flight),* refund *(customer is refunded for the ticket),* dateChange *(the flight is rescheduled).*

The temporal behavior of an airfare that allows flight reschedulings (e.g. ***Ticket A*** *of Example 2) is described by clauses satisfied by temporal sequences in which this happens, but also sequences in which the ticket is used in the originally scheduled date:*
1. *(purchase) ⇢ (use) ⇢ ...*
2. *(purchase) ⇢ (dateChange) ⇢ (use) ⇢ ...*
3. *...*

A customer looking for a ticket that allows reschedules even after a missed flight, would look for sequences with the form:
(purchase) ⇢ (missedFlight) ⇢ (dateChange) ⇢ ...

We believe that representing contracts' temporal behavior and user queries with declarative clauses leads to a system that satisfies the aforementioned requirements. The use of a common vocabulary along with ad hoc declarative clauses satisfies both the need of expressiveness and the requirement of a compact and stable interface (*i* and *ii*). In §2.1 we will show how with our notion of permission, we avoid unnecessary revisions of contracts specifications (*iii*). Finally, we show in §2.2 that our system representation is reasonably close to the natural language specification of Example 2 (*iv*).

Summarizing our problem setting:
a. We focus on the temporal aspect of querying contracts, so we assume a traditional DBMS takes care of the features modeled as relational attributes.
b. The temporal behavior of a *contract* is *specified with declarative clauses* that represent a *set of allowed temporal sequences of events*.
d. A contract *permits* a temporal query (also specified as a declarative clause) if it allows a sequence satisfying the query property.
e. Our system stores the temporal specification of all contracts and has to provide a scalable and high performance way to retrieve all contracts that permit an online temporal query.

Solving this problem involved the following contributions:
1. We model an intuitive notion of permission with respect to contracts and queries, which captures the temporal behavior aspects that are not handled currently by state-of-the-art systems. Our notion gracefully handles the fact that contracts might not specify the behavior of all events mentioned in a query, finding the best design tradeoff between two extremes. Indeed, returning contracts that do not *explicitly* allow the requested behavior for a particular event would incentivize publishing underspecified contracts, making the system useless to consumers. While forcing full specification would dramatically increase publishing costs.
2. Aiming to represent and reason about temporal aspects, we picked a standard and well established formalism, namely Linear Temporal Logic. Checking permission does not reduce to any of the many decision problems studied for LTL, so we developed a novel algorithm. Note that we do not expect end users to utilize LTL directly. Analogously to SQL in database-powered e-commerce systems, LTL will be used in our system only by application developers. User-friendly GUIs for LTL have previously been studied [5], but they are not the focus of this paper.
3. We develop a novel indexing scheme for large scale databases of contracts that allows our system to quickly identify a set of candidate contracts for a given temporal query. This eliminates the need to execute the complex permission algorithm on every single contract temporal specification in the repository and is extremely effective for highly selective complex queries.

4. We develop a novel way to automatically simplify temporal specifications of contracts based on the events cited in a query. This allows our system to execute the permission algorithm on smaller contract specifications and provides the best results for simple queries that mention few events.
5. Finally, we provide an experimental evaluation of our techniques to prove the feasibility of our approach. Exploiting both our indexing techniques, we show an optimized average query evaluation time of 6sec over databases of 3000 contracts, improving by more than an order of magnitude over the unoptimized average time of 98sec. Note that our prototype ran on a standard desktop pc and did not take advantage of multiple cores, as we focused on the benefits of the indexing techniques and not of a particular hardware platform. The size of the tested databases is adequate for most applications as a complete implementation would use relational attributes (e.g. travel date and destination) to pre-select contracts from a potentially much larger database.

The rest of the paper is organized as follows. Section 2 informally introduces the temporal search problem, Section 3 presents a high level unoptimized system architecture along with our novel permission algorithm, Section 4 describes our indexing scheme, Section 5 describes our automatic simplification technique, Section 6 details the formal foundations of our work, Section 7 presents our experimental results, Section 8 discusses some related work, and Section 9 concludes the paper.

2. QUERYING CONTRACTS

To ease the presentation of our contributions we provide an intuitive description of our work in the first sections of the paper and we relegate the formal foundations of our results to §6.

Following §1, we refine the intuition of 'sequence of events' by adding the notion of *snapshot*. At every moment in the temporal sequence, many events could be happening, a *snapshot* captures all the information regarding the events in a particular moment (i.e. a truth assignment for every event in the vocabulary). A temporal *sequence* is then just a list of snapshots. Intuitively, we are able to define declaratively properties of these temporal sequences using *clauses*. We define a *contract* as a set of clauses that identify the set of allowed temporal sequences of snapshots, and a *query* as a clause that specifies the required property of a temporal sequence.

2.1 Designing the Permission Semantics

Usually, customers are interested in knowing if a certain sequence will be possible if they subscribe to a contract. More precisely, they are interested in a particular property of that sequence (e.g. the fact that a ticket is partially refundable), which they specify using the query clause. A natural first-cut semantics would be that a contract is returned as a result of a query q, if at least one of its allowed sequences satisfies q. This semantics, however, has a subtle problem.

EXAMPLE 4. *Let us consider **Ticket A** of Example 2, but with a common vocabulary that contains also the event* 'class upgrade'. *The airfare policy is still that no refund is allowed after a date change and that date changes are unlimited. Let us assume that a customer issues the following query:*

Q2 *An airfare that allows a class upgrade after a date change.*

*If we think of a sequence of events in which the customer obtains a date change and then obtains a class upgrade, we can easily see that it satisfies all clauses of **Ticket A** (since they impose no restriction of class upgrades, which are not mentioned). It follows that under this semantics **Ticket A** will be returned as part of the result.*

In Example 4, using the first-cut semantics, a contract, which did not say anything about class upgrades, is returned as part of the result of a query regarding class upgrades. Clearly, the contract is not fully specified with respect to **Q2**. We contend that returning contracts that are underspecified w.r.t. a query (as in Example 4) would not serve customers well for two reasons. First, customers would be required to read all returned contracts in order to know which contracts explicitly permit the query and which ones were underspecified. Second, in order to gain visibility, publishers would be incentivized to publish underspecied contracts, that would be selected by more queries. This would exacerbate the first problem and make our system useless. However, we cannot solve the problem of underspecified contracts by forcing full specification on the part of contract publishers, as it would raise the publishing cost considerably. It would also prevent a simple introduction of new events in the common vocabulary, as a modified vocabulary would force a revision of all contract specifications in the system.

For these reasons we refine the first-cut semantics in order to exclude contracts that do not explicitly allow some events. Our refined semantics takes into account only the events cited in each contract, and assumes that the contract makes no commitment on events not explicitly cited in its clauses. In Example 4, **Ticket A** would not be returned because it never cites events regarding class upgrades. This discussion motivates the following final semantics for permission, stated formally in §6.1 and informally as:

Definition 1. A contract *permits* a query if and only if there exists a sequence that **(a)** is allowed by the contract, **(b)** consists only of the events mentioned in the contract, and **(c)** satisfies the query.

Note that the restriction to the events cited in the contract does not result simply in the immediate conclusion that a contract that does not mention all events in a query q does not permit q. Considering **Ticket B** of Example 2 (i.e. all refunds, date changes only before scheduled departure), it is easy to see that it should be returned if the customer is interested in the following contracts:

Q3 After a date change, airfare allows a class upgrade or a refund.

This is because, even though **Ticket B** does not specify a class upgrade policy, it explicitly allows refunds after date changes.

2.2 Declarative language

During the modeling of both synthetic and real-world contracts and queries, we found that temporal logic resulted expressive enough to capture the temporal behaviors at hand and also very close to the informal specification. Moreover, many other domains deal with temporal behavior of systems, most notably model checking and verification ([25]). In all these domains, temporal logic has become the accepted standard formalism for declaration of properties over temporal sequences, and many techniques based on this kind of logic are successfully applied to industrial level implementations of tools [14], [4], [17].

We use the simplest form of temporal logic (Linear Temporal Logic, LTL [20]) as the formalism to capture declarative clauses, informally introduced in §1. For the formal treatment of the use of LTL refer to §6.1. As an example, we report here the LTL specification of **Ticket C** from Example 2:

Ticket C
1. $\mathbf{G}(\neg refund)$
2. $\mathbf{G}(dateChange \rightarrow \mathbf{X}(\neg \mathbf{F} dateChange))$
3. $\mathbf{G}(missedFlight \rightarrow \neg \mathbf{F} dateChange)$

Clause 1 is read 'globally not *refund*', *globally* (**G**) means that a certain formula has to be true in all the snapshots of a sequence. Clause 2 reads 'globally, if a date change occurs, then in the next instant (**X**) it is not the case that eventually another date change occurs', *eventually* (**F**) means that a certain formula will be true in some future instant. Clause 3 reads 'globally, if a flight is missed, it is never the case that eventually a date chage occurs'. Note how these formulas are close to the informal description of Example 2.

In our LTL specifications, we consider a common vocabulary of propositional variables $\mathcal{V}$. A variable is intuitively associated to an event and it is $true$ in a snapshot where its associated event happens. LTL is essentially a propositional logic which, in addition to the usual boolean operators, uses some temporal operators with the following intuitive meaning:

- **X**p, p is true on the *next* instant
- **F**p, *eventually* p is true
- p**U**q, q is eventually true and p is true *until* q is true
- **G**p, *globally* (i.e. in every instant) p is true
- p**B**q, p is true *before* q is true
- p**W**q, *weak until*: p is always true or p is true *until* q is true

EXAMPLE 5. *Let us see how the clauses of the contracts in Example 2 can be expressed in LTL. We assume a vocabulary $\mathcal{V}$ containing the event variables* {purchase, use, missedFlight, refund, dateChange}.*First, we need to model the intuitive meaning of the events in the vocabulary with some additional clauses that are common to all three airfares. Informally:*

C1 *The ticket is purchased once.*

C2 *The ticket has to be purchased before any of the following events: it is used, the flight missed or rescheduled, or the customer is refunded.*

C3 *If the flight is missed, the ticket is unusable unless rescheduled*

C4 *If a refund is issued, then no other event can happen*

C5 *If the ticket is used, then no other event can happen*

The full LTL specification is then:

C0 *In this case, we force the fact that only one event happens in each instant,*
$\boldsymbol{G}(purchase \rightarrow \neg use \wedge \neg missedFlight \wedge \neg refund \wedge \neg dateChange)$,
$\boldsymbol{G}(use \rightarrow \neg purchase \wedge \neg missedFlight \wedge \neg refund \wedge \neg dateChange)$,
...,

C1 $\boldsymbol{G}(purchase \rightarrow \boldsymbol{X}(\neg \boldsymbol{F} purchase))$

C2 $\boldsymbol{G}(purchase\ \boldsymbol{B}\ (use \vee missedFlight \vee refund \vee dateChange))$

C3 $\boldsymbol{G}((missedFlight \rightarrow \neg \boldsymbol{F} use)\ \boldsymbol{W}\ dateChange)$

C4 $\boldsymbol{G}(refund \rightarrow \neg \boldsymbol{F}(use \vee missedFlight \vee refund \vee dateChange))$

C5 $\boldsymbol{G}(use \rightarrow \neg \boldsymbol{F}(use \vee missedFlight \vee refund \vee dateChange))$

Ticket A *1.* $\boldsymbol{G}(dateChange \rightarrow \neg \boldsymbol{F} refund)$
2. no clause necessary

Ticket B *1.* no clause necessary
2. $\boldsymbol{G}(missedFlight \rightarrow \neg \boldsymbol{F} dateChange)$

Ticket C *1.* $\boldsymbol{G}(\neg refund)$
2. $\boldsymbol{G}(dateChange \rightarrow \boldsymbol{X}(\neg \boldsymbol{F} dateChange))$
3. $\boldsymbol{G}(missedFlight \rightarrow \neg \boldsymbol{F} dateChange)$

*Note that the specifications of the contracts will be the conjunction of all common clauses along with the specific ones (e.g. for **Ticket A**: C0,C1,C2,C3,C4,C5,TicketA). Also, note that some natural language clauses (e.g. 'Unlimited date changes') did not require any additional LTL clause, as they are implicit in the LTL semantics. This is because they did not add any additional constraints to what was already specified in the common clauses (e.g. for date changes: C0,C2,C3,C4,C5).*

LTL as a developer language We want to stress the fact that the usage of LTL in our scenario is similar to the use of SQL in database powered e-commerce applications: LTL will be used by the application developers. User-friendly GUIs for LTL have been studied [5] and they are not the focus of the present paper.

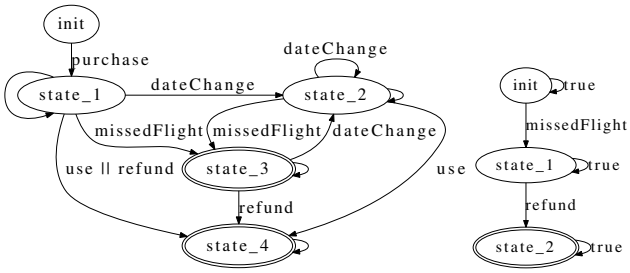


Figure 1: BAs for: (a) **Ticket A** of Example 2 (no refund after a date change AND unlimited date changes, (b) a query asking for a refund after a missed flight. Double circled states are final. **For readability:** In (a), we assume that every label is a conjunction containing a literal for every event mentioned in the contract. We show only the positive ones, e.g. $refund$ is short for $refund \wedge \neg purchase \wedge \neg missedFlight \wedge \neg dateChange \wedge \neg use$. No label means that all the events are negated.

2.3 Data model

In order to reason on temporal clauses expressed in LTL, we make use of an important connection [28] between LTL and a particular kind of finite state automata, called Büchi automata (BA). Exploiting results in [28], many tools (e.g. [12]) build a BA accepting all and only the sequences that satisfy a given set of declarative LTL clauses. As we will see, this reduces our problem to searching a database of BAs representing contracts, while checking an individual contract reduces to exploring paths in its BA.

Büchi automata (BA) are similar to standard finite state automata, with the exception that they recognize infinite strings. The BA acceptance condition is that a string is accepted if the recognizing automaton traverses a final state infinitely many times. Note that the labels of our BAs are conjunctions of literals (i.e. conditions postulating the existence (positive) or absence (negated) of a single event in a snapshot, e.g. $purchase$, $\neg use$) that have to be satisfied by the current snapshot, in order for the transition to be enabled during recognition (Figure 1).

To some readers the choice of using infinite sequences might seem unnecessarily complicated. We note, however, that using infinite sequences does not reduce the expressive power of the formalism, as any finite sequence can be encoded simply appending an infinite list of dummy snapshots. Moreover, for technical reasons, infinite sequences are the standard formalization when reasoning about temporal properties [28].

We provide now an intuitive example of BAs in our context (the formal definitions can be found in §6.2.1).

EXAMPLE 6. *Let us consider the Büchi automaton in figure 1b. If we use the automaton to recognize a sequence of snapshots, it is easy to check that it will accept it if and only if this sequence contains a snapshot in which $missedFlight$ is true and in at least one snapshot after that $refund$ is true. Indeed, recall that the accepting condition is that the automaton traverses $state\_2$ infinitely many times. Since this state has a self loop with $true$ as label, it means that once that state is reached, it will always be part of a cycle, and thus traversed infinitely many times.*

3. SYSTEM ARCHITECTURE

In §3.1 we introduce an algorithm that checks permission of a query by a contract using their BA representations. This algorithm suggests a straightforward implementation for our brokering system. At registration time, a contract LTL specification is translated to an equivalent BA representation that is the one stored. In our prototype we use an existing tool [12] to perform this stage. At query time, the query in LTL form is converted to an equivalent

BA in the same way and then checked (using the above mentioned algorithm) against all the contract BAs in the database. As we will see in §3.1, however, the permission problem has a high complexity lower bound, which makes such an approach impractical.

We expect our contract database to be fairly static and each contract to be queried multiple times. Our indexing techniques (§4 and §5) share the common idea of precomputing some auxiliary information during the registration step, in order to achieve faster online query processing. In essence, we are indexing a set of BAs for permission lookup.

3.1 Algorithm

Our algorithm uses the BA representations of a contract and a query to check if the first permits the second. From the BA accepting condition in §2.3 and Example 6, it follows immediately that, to accept a sequence, there is a path through the states of a recognizing BA that is formed by concatenating a finite simple path (*prefix*) that leads to a final state k (called *knot*) and by iterating infinitely many times through a simple *cycle* that leads back to k. These kind of paths are called *lasso paths*. They are infinite but have a finite representation, consisting of the prefix and the cycle.

A classical result [28] states that a sequence ρ satisfies a temporal property φ if and only if any BA for φ traverses a lasso path while recognizing ρ. We refer again to §6.2.1 for a formal treatment of BA in our context.

Now recall Definition 1 in §2.1. Assuming that both the contract and the query are represented by the BAs of all their allowed sequences, we provide below the intuition that the lasso path concept can be exploited for a permission checking algorithm.

EXAMPLE 7. *Let us consider the contract BA in figure 1a and the query in figure 1b. It is easy to see that the sequences that are accepted by both automata satisfy a lasso path of the form* $\{init,$ 1, ..., 1, 3, ..., 3, 4, 4, ... $\}$ *in the contract BA, and of the form* $\{init,$ .., $init,$ 1, ..., 1, 2, 2, ... $\}$ *in the query BA.*

Intuitively, we see that we can build a sequence that satisfies a lasso path for both the contract BA and the query BA (i.e. we just need to synchronize the traversal of the transitions $1 \rightarrow 3$ *in the contract BA with the* $init \rightarrow 1$ *in the query BA and* $3 \rightarrow 4$ *with* $1 \rightarrow 2$*). Note that in this process we tried to pair the labels on the mentioned transitions (evidently not all labels are 'compatible').*

The intuition in Example 7 rests on the concept of pairing a lasso path on the contract BA with another lasso path on the query BA. We can provide a more precise intuition by introducing the concept of *simultaneous lasso path*.

Definition 2. A *simultaneous lasso path* is a sequence of pairs of states $\langle s_i, q_i \rangle_{i \geq 0}$ so that:

1. $\langle s_i \rangle_{i \geq 0}$ is a lasso path in the contract BA
2. $\langle q_i \rangle_{i \geq 0}$ is a lasso path in the query BA
3. for every i, the label α_i of the i^{th} transition of $\langle q_i \rangle_{i \geq 0}$ and the label σ_i of the i^{th} transition of $\langle s_i \rangle_{i \geq 0}$ are *compatible*. This happens when: (i) α_i refers only to events in the contract, and (ii) α_i and σ_i do not conflict (i.e. if α_i contains *dateChange*, σ_i cannot contain $\neg$*dateChange*)

In §6.2.1 we formally treat this intuition, resulting in:

THEOREM 1. *Let C be a contract and q a query. There exists a simultaneous-lasso path in the BAs representing C and q if and only if C permits q.*

Note that simultaneous lasso paths as in Definition 2 are infinite. However, we can finitely represent them with the prefixes and cycles for both the contract and query lasso paths. Given a BA, note that there exists a finite number of simple paths that can be used as prefixes and cycles. This provides us with a finite search space for all possible lasso path representations. It follows that, using Theorem 1, we can naively solve the permission problem by generating all possible representations of lasso paths for the query and contract BAs and then checking if they are a simultaneous lasso path (i.e. they satisfy Definition 2). This naive algorithm implies decidability of the permission problem.

A better algorithm can be coded with a simple backtracking search on the two BAs. We refer to §6.2 for the complete description of the algorithm. However, the details provided in this section suffice to follow the presentation of our indexing techniques.

Complexity The backtracking search results in a LOGSPACE complexity in the size of the BAs. The size of BAs, however, is known to be worst case exponential in the size of the original clauses (i.e. contract specifications and queries) [28]. This results in a PSPACE upper bound. With a reduction from LTL satisfiability [22] we prove in [6] that the bound is tight and the problem is indeed PSPACE-complete. While this complexity implies an exponential running time, it is not an impediment to a practical implementation, for two reasons. First, the exponentiality is in the size of the query and a single contract, both of which are relatively small compared to the size of the contract database. Second, the translation from clauses to BA seldom results in an exponential blowup. Indeed, our experiments confirm that a practical implementation is feasible due to the indexing techniques described next.

4. PREFILTERING INDEX

The technique that we present in this section reduces the number of executions of the verification algorithm by pruning the number of contracts to test. At registration time, an index data structure is updated with information about the contract. At query time, the index is used to identify a set of contracts which is guaranteed to contain all contracts permitting the query. The permission algorithm can then run only on these contracts (*candidates*).

At a high level, the technique extracts a necessary condition for permission from the query BA (called *pruning condition*), it then uses the index data structure to quickly identify the set of candidate contracts satisfying that condition.

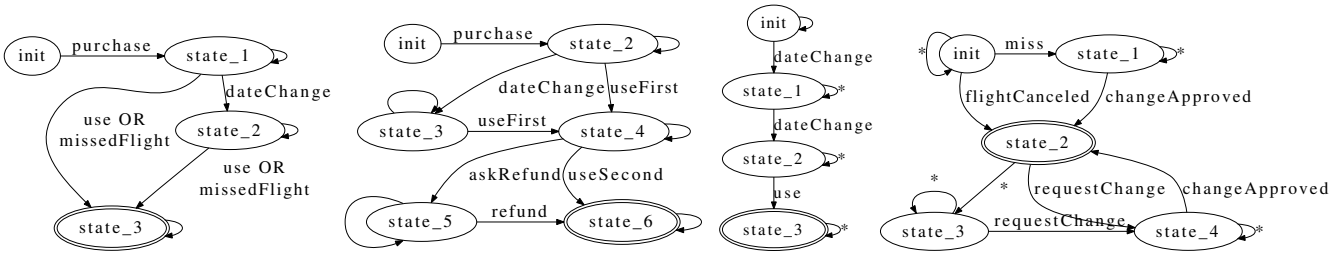
4.1 Pruning Conditions

EXAMPLE 8. *Consider the query BA in Figure 2c. Since permission is checked looking for a simultaneous lasso path, it is clear that, in order to permit this query, a contract BA must have some transitions that are compatible (as in Definition 2, point 3) with both formulas:* $dateChange$ *and* use*. Otherwise, no simultaneous lasso path can be built. While this condition is necessary, it is clearly not sufficient: the contract in Figure 2a has such transitions but does not permit the stated query.*

We now generalize the idea of Example 8. From Theorem 1 we know that we have to build a simultaneous lasso path. This means that, by definition, a lasso path has to exist in the query BA. We also know that all transitions in this query lasso path have to be 'compatible' to their simultaneous transitions in the contract BA. It follows that in the contract BA, at the very least, there has to exist one compatible label for every label in the query lasso path.

From this observation, it follows that if we find all lasso paths in the query BA, we can create a set of candidates by gathering all contracts whose BAs contain a compatible label for every label of a query lasso path. The problem is that the number of query lasso paths is exponential in the size of the query automaton. However, we never explicitly compute all lasso paths, as we can extract a more compact representation directly from the query BA.

In order to explain this, we assume access to a data structure that returns $S(\lambda)$, the set of contracts in the repository that contain a label compatible with a given query BA label λ. We will describe



(a) (b) (c) (d)

Figure 2: Büchi automata for (a) a ticket that allows no refunds and date changes only before the scheduled departure (**Ticket C** of Example 2), (b) a round-trip ticket allowing a single change before the first flight and a refund for the second one, (c) a query asking for two date changes, (d) a query asking for tickets that can be changed indefinitely even after a flight is canceled or it is missed and already rescheduled once. Double circled states are final. **Note**: BA (a) and (b) follow the same convention as Figure 1a.

the implementation of this data structure in §4.2.

EXAMPLE 9. *Consider the query BA in Figure 2d. It only contains a single final state* `state_2`*. In order to build a lasso path knotted in* `state_2`*, we need to have both a prefix and a cycle. By visual inspection, there are only two possible prefixes that are simple paths (i.e. do not contain cycles): one with label* $flightCanceled$*, the other with* $miss$ *and* $changeApproved$*. We do not consider the self-loops in* `init` *and* `state_1` *(and any other cycle) because their labels are not strictly necessary to build a prefix, so we cannot exclude any contract for not having them. Knowing all the possible prefixes, it follows that all the contracts that contain labels compatible with these prefixes are in the following set:*

$S(flightCancelled) \cup (S(miss) \cap S(changeApproved))$

The idea is that this set will be smaller than the whole database. We can also prune other contracts by considering the cycles of the query lasso paths knotted in `state_2`*. Since the transition from* `state_2` *to* `state_3` *is labeled with* $true$*, it provides no way to prune any service. It follows that in order to go from* `state_2` *to* `state_4` *we can only force the presence of a transition compatible with* $requestChange$*. In order to reach back* `state_2` *we need* $changeApproved$*. Adding this condition to the previous one (i.e. intersecting the two sets), we know that we need to run the permission algorithm only for contracts in the following set:*

$(S(flightCancelled) \cup (S(miss) \cap S(changeApproved))) \cap (S(requestChange) \cap S(changeApproved))$ □

In Example 9 we generated a set of candidates from a condition derived from the analysis of lasso paths knotted in `state_2`. We call conditions of this kind *lasso pruning conditions*. A lasso pruning condition for a knot k selects all contracts that permit the query with a simultaneous lasso path whose query lasso is knotted in k.

It is clear that any contract permitting the query in Example 9 will permit it with a simultaneous lasso path whose query lasso is knotted in `state_2`. This is because the query BA has a single final state. In general, a query BA contains more than one final state.Since any simultaneous lasso path is enough to guarantee permission, we have to build a lasso pruning condition for every final state and then take the union of the sets retrieved by them.

Our implementation, fully described in [6], exploits a memoization scheme to compute lasso pruning conditions, that results in a linear complexity w.r.t. the query BA size.

4.2 Index data structure

In §4.1, we assumed a way to easily compute $S(\lambda)$, i.e. the set of contracts in the repository that contain a label compatible with a query BA label λ. We will now introduce the index data structure that we use to perform this operation.

At a high level, we have to find a way to identify labels that are compatible with λ, and then gather all contracts containing them. Clearly, we do not want to scan the set of all labels in the contract database. The main idea is to use the literals of λ to navigate a data structure that contains sets of contracts.

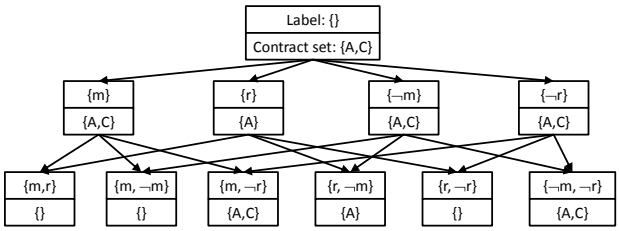


Figure 3: Prefilter index for the two contracts in Figure 1a (**Ticket A**, called A) and 2a (**Ticket C**, called C). For readability we only show nodes with labels containing events *missedFlight* (*m*) and *refund* (*r*) and the structure is limited to the first two levels.

Let us assume, for ease of presentation, that each label in contract BAs contains all events in the contract vocabulary (we shall drop this assumption shortly). This means that checking compatibility can be peformed simply by checking that the λ literals are included in the contract labels' literals. Since in the labels the order of literals is not important, the problem reduces to searching in a collection of sets for superstes of a query set. For this purpose we modify a standard TRIE [11] data structure by making it a simple directed acyclic graph (Figure 3). The root is connected with nodes labeled with single literals. Every node labeled with a literal l in this first 'level' is connected with nodes in a second level that are labeled with a set of two literals containing l, and so on. Clearly, this structure is not a tree because, for instance, a node in the second level labeled $\{a, \neg b\}$ is reachable from the node $\{a\}$ and the node $\{\neg b\}$. We, also, associate each node labeled with the set of literals l to the set of contracts whose BAs have at least one transition with a label containing l. Now, in order to retrieve $S(\lambda)$, we just navigate the graph to the node labeled with the literals in λ, and output its associated set of contracts. This lookup is linear in the number of literals in λ and independent of the number of contracts.

EXAMPLE 10. *Consider the two contracts in Figure 1a (**Ticket A**) and 2a (**Ticket C**), called A and C. For readability, we show only a partial view of the full data structure in Figure 3 (only events* $missedFlight$ *and* $refund$*, only the first two levels). Remember that the BAs in Figures 1a and 2a show only the positive literals, as noted in Figure 1a. So, for instance, both BAs have a transition labeled* $missedFlight$ *in the pictures, that actually stands for* $missedFlight \wedge \neg refund$*, considering for readability only the aforementioned events. This results in A and C being added to the nodes labeled {*$missedFlight$*,*$refund$*}, {*$missedFlight$*} and {*$refund$*}. Note how only **Ticket A** has a label with* $refund$*.*

If we consider the query in Figure 1b, it is easy to see that a lasso pruning condition is $S(m) \cap S(r)$. Trivially, $S(m)$ is the set

of contracts associated with the node labeled m and evaluates to $\{A, C\}$. $S(r)$ is $\{A\}$. The whole condition selects the candidate $\{A\}$. Using the prefiltering technique, we avoid the execution of the permission checking algorithm on contract C.

In Example 10, the transitions of contract BAs always contain all events mentioned in the contracts. This allowed us to use our previous assumption and check compatibility using containment. This is not always the case, but we present now a way to reduce compatibility to containment by introducing additional literals to the nodes' labels. First, we note that the reason our previous assumption does not work in general is because, in order to check compatibility between λ and γ, we might have to consider not only the literals in γ but also the ones cited in its contract c. This is the case, for instance, if γ is the boolean formula $refund$, in a contract that cites both $refund$ and $dateChange$. We have then that both labels, $refund \wedge dateChange$ and $refund \wedge \neg dateChange$ are compatible with γ. In order to treat compatibility checks as containment checks, we create a set for γ (called the *expansion*, $\mathcal{E}(\gamma)$) that contains all literals in γ plus all literals (both positive and negative) for any remaining event cited in its contract.

EXAMPLE 11. *Let us consider a transition label $t = p \wedge c$ from a contract that cites the events p, c and m. Given the above definition we have that its expansion $\mathcal{E}(p \wedge c)$ = $\{p, c, m, \neg m\}$. We have that a query transition $q = p \wedge m$ is compatible with t, because $\{p, m\} \in \mathcal{E}(p \wedge c)$. And $q' = p \wedge \neg c$ and $q'' = c \wedge r$ are not, because $\{p, \neg c\} \notin \mathcal{E}(p \wedge c)$ and $\{c, r\} \notin \mathcal{E}(p \wedge c)$.*

Now, we can adopt the same strategy shown in Example 10 just by changing the contract set associated to each node. Every node labeled with a set of literals l is now associated with the set of contracts whose BAs have at least a transition with label γ *s.t.* $\mathcal{E}(\gamma)$ *contains* l.

The problem we face in the implementation and scaling of this solution is that the number of possible node labels is exponential in the size of the vocabulary. This means that the index grows exponentially in size, leading to intractability. In order to avoid this exponential blow up, we limit the size of node labels (i.e. the number of 'levels' of the data structure) to a predefined number k. We now have to deal with the lack of nodes with labels greater than k, as nothing changes when retrieving $S(\lambda)$ with $|\lambda| \leq k$. On the other hand, when we retrieve $S(\lambda)$ with $|\lambda| > k$, we can just take any set associated with a node whose label l *is contained in* the literals of λ. We call this set $S'(\lambda)$ and, by construction, $S'(\lambda)$ contains $S(\lambda)$.

This modification does not affect the soundness and completeness of the technique because pruning conditions need to evaluate to a set that just *contains* a certain set of contracts (§4.1). Since conditions use only union and intersection operators, they are monotonic. This means that the fact that the $S'()$ returned by the data structure are supersets of the required $S()$ results in a set of candidates that is a superset of the one returned if we were using the correct $S()$. Since all candidates will be checked with our permission algorithm and we are returning a superset of the original set of candidates, soundness and completeness are preserved.

5. BISIMULATION INDEX

In this section we describe an indexing technique that is based on the observation that the query usually pertains to a much smaller set of events than the full contract specification. At registration time, we precompute a set of simplified versions of every contract BA. Intuitively, these simplified versions can be thought as 'projections' of the full contract specification on a subset of events (i.e. albeit smaller, they faithfully represent the behavior of the contract with respect only to the considered events). At query time, our system retrieves the appropriate precomputed 'projection' for each candidate contract that has to be checked for permission.

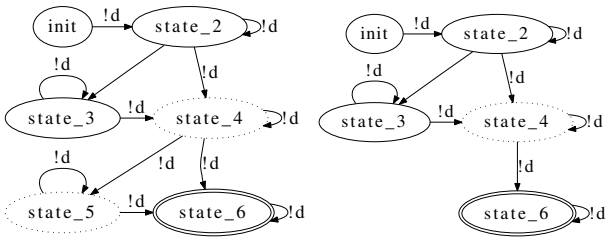


Figure 4: The dotted states can be substituted with a single one, when checking permission w.r.t. query regarding only *dateChange*. For readability: *d* = *dateChange*. (a) projection of BA in Figure 2b on ¬ *dateChange*. (b) simplified version of (a).

5.1 Simplified contract automaton

We will show now how we use a simplified version in lieu of the full contract BA.First we define a notion of equivalence with respect to a query q: two contracts BA $\mathcal{A}$ and $\mathcal{B}$ are *equivalent* if they both either permit or do not permit q. From Theorem 1, it follows that this is the case if and only if we can build simultaneous lasso paths with the query BA for $\mathcal{A}$ and $\mathcal{B}$, or we fail for both.

We will now show how to simplify the transitions of a contract BA in order to obtain a *projection* BA that is equivalent to the original one w.r.t. a query. Then, we will reduce the number of states of this projection obtaining a *simplified* BA, on which to run our permission algorithm.

Let us start by simplifying the transitions. First, remember that in order to build a simultaneous lasso path we need two lasso paths (one for the contract BA, one for the query BA) whose transitions are compatible in every instant. It follows that the only information needed to find this simultaneous lasso path is if a contract label is compatible with a query label. For instance, if a query BA has transitions that contain only the labels $true$ and $dateChange$, the only information we need to derive compatibility (besides knowing all events in the contract) is the presence of $\neg dateChange$ in the contract transitions. It should be intuitive now why it is useful to introduce the concept of projection. Given a contract BA $\mathcal{A}$ and a set of literals L, we call *projection* of $\mathcal{A}$ on L (i.e. $\pi_L(\mathcal{A})$) the BA built from $\mathcal{A}$ by keeping in every label only literals in L, e.g. Figure 4a represents the projection BA on $\neg dateChange$ of the BA in Figure 2b.

In Theorem 5, §6.3, we formalize the following result:

THEOREM 2. *Given a BA $\mathcal{A}$ and a set of literals L, the projection of $\mathcal{A}$ on L is equivalent to $\mathcal{A}$ for all query BAs that refer to literals that are either negations of literals in L or pertain to events not in cited in $\mathcal{A}$.*

The use of projections alone does not improve the performance of the permission checking algorithm as the size of the projection BA is the same as the original one. Projections are, however, simpler BAs than their original counterparts in the sense that many transitions that had different labels have now the same label.

This provides the opportunity to use a standard state reduction technique [10] that produces a BA that is equivalent to the original one (i.e. it accepts the same set of strings). This classical polynomial-time technique is based on *collapsing* bisimilar states (i.e. replacing them with their bisimilarity class) and is the same used for the minimization of standard finite state automata [15]. In the case of BAs it does not produce the minimum BA (BA mini-

mization is PSPACE-hard [23]), it does provide, however, significant reductions in the number of states of our projections. This allows us to equivalently check our query on a contract BA that is a projection of our original one but that has significatly fewer states.

EXAMPLE 12. *In Figure 4a we show a BA derived from a projection, where we have two states (*`state_4` *and* `state_5`*) that can be collapsed. Intuitively, once the recognizing BA reaches either of those states, it will accept* the exact same set of sequences*, i.e. an infinite sequence of snapshots containing* $\neg dateChange$*. Its simplification is shown in 4b.*

5.2 Using simplified BAs

The previous section details how we can equivalently use simplified projections to check permission. As we already anticipated at the beginning of the section, for every contract we precompute the projections at registration time, so that when our system has to check permission of a query, it can use the precomputed simplified projection. Clearly, the most appropriate projection is the smallest one that is still equivalent to the original BA for the current query.

Given the result in Theorem 2, it is easy to see that in order to have the smallest projection for every possible query, we need to have one projection for every subset of literals cited in the contract. The problem that immediately arises is the fact that the number of possible projections is, in the worst case, exponential in the size of the set of events of the contract. We have, however, two important observations that enable us to pursue the precomputation route.

The first observation is that we do not need to precompute all projections, as we can use *any* projection that contains all required literals for the query, albeit with a possible perfomance penalty. The second observation is that, in practice, not all subsets of literals generate distinct simplified projection BAs, e.g. in our datasets the number of distinct simplified projection BAs was ∼5% of the number of subsets. Intuitively, removing a single event from a projection does not usually lead to a simplification of the observed behavior of the others. It follows that it is often the case that the simplifications are triggered only when whole sets of 'independent' events are excluded from the projection. Both these observations provide us great freedom in designing a system that balances runtime performance with precomputation time and storage needs.

Storage needs are quite contained, since we do not need to store a graph for every projection. Since we have the original contract BA, we can just memorize the list of bisimilar states for a particular projection. With some care in the implementation, the permission algorithm does not incur any substantial overhead.

To address precomputation time, we designed a novel algorithm that efficiently computes all simplifications in parallel. We could not use the standard bisimulation algorithm ([19]), as we would have had to run it for every subset of literals. Our algorithm avoids testing subsets that are not going to result in simplifications and reuses partial computations between different subsets. For space reasons, we relegate the description of this algorithm to [6].

In our experiments we found it feasible to precompute all projections. In the case that contract complexity grows in a way that precludes full precomputation, we can limit it to subsets of literals up to a certain size k. This would not affect the evaluation perfomance of queries with less or exactly k literals, which are the ones that mostly benefit from this technique, as the contract BAs gets significantly simplified. It would affect the evaluation performance of queries with more than k literals, which, however, mostly benefit from the complementary prefiltering index. Other possible approaches to address increasing precomputation times include the use of heuristics based on historical data and/or expected workloads to determine which simplification to precompute.

6. FORMAL TREATMENT

6.1 Permission Problem

The formal counterpart of our intuitive notion of temporal sequence is the *run*. A *run* ρ is a function $\rho(t) : \mathbb{N} \rightarrow T(\mathcal{V})$, where $\mathbb{N}$ are the natural numbers, $T(\mathcal{V})$ is the set of truth assignments for the event variables in $\mathcal{V}$, and t a particular instant. The 'tail' of the run starting at instant i is written as $\rho|_i$.

The semantics of LTL is given inductively for all boolean operators and for the temporal operators (**X**, **U**). Note that all temporal operators can be derived from **X** and **U** (e.g. $\mathbf{F}\varphi \equiv true\ \mathbf{U}\ \varphi$, $\mathbf{G}\varphi \equiv \neg\mathbf{F}(\neg\varphi)$, $\varphi\mathbf{B}\psi \equiv \neg(\neg\varphi\ \mathbf{U}\ \psi)$). A run ρ *satisfies* an LTL formula φ (i.e. $\rho \models \varphi$), iff, inductively:

- $\rho \models p$, with $p \in \mathcal{V}$, iff p is true in $\rho(0)$
- $\rho \models \mathbf{X}\varphi$, iff $\rho|1 \models \varphi$
- $\rho \models \phi\mathbf{U}\psi$, iff $\exists k \geq 0$ s.t. $\rho|k \models \psi$ and $\forall 0 \leq i < k(\rho|i \models \phi)$
- closure w.r.t. boolean operators ($\wedge$, $\neg$)

Remember from the discussion in §2.1 that our intuitive semantics did not take into account the behavior of events that were not explicitly cited in the contract. In particular, we wanted to avoid cases where the behavior of a variable not cited in the contract specification results in the contract permission of a query that is not intended to be supported by the contract. In order to formalize this intuition we introduce the following concepts.

Definition 3. The *projection* of a run ρ w.r.t a set of event variables V (a V-projection), is a sequence σ that assigns to the variables in V the same truth values assigned by ρ, and does not assign any truth value to other variables. A run ρ is *compatible* with a V-projection σ, if σ assigns in every instant the same truth values of ρ to all variables in V. The set of all runs compatible with a V-projection is called *projection class*.

Definition 4. Given an LTL formula φ, let V be the set of its variables. We call $P(\varphi)$ (*set of allowed projections of a contract* specified by φ) the set of V-projections of all runs satisfying φ.

Definition 5. Given an LTL formula φ, let V be the set of its variables. The contract defined by φ (referred to as $C(\varphi)$) *permits* an LTL query ψ iff there exists a V-projection $\sigma \in P(\varphi)$ s.t. *all* runs in its projection class satisfy ψ.

Intuitively, forcing the property to be true for all runs in a projection class avoids the case when a run triggers permission only due to truth assignments of variables not mentioned in the contract.

EXAMPLE 13. *Consider a query q that is satisfied by a run ρ of a contract only because of the behavior of variable v. This means that without the specific behavior of the variable v, q would not be permitted, as is the case in Example 4 for the variable 'class upgrade'. Note that in the projection class of ρ there is a run ρ′ that does not exhibit the same behavior for v, as v is not part of the contract vocabulary. It follows that q is not satisfied by all runs in a projection class and thus it is not permitted by the contract.*

6.2 Checking permission

It is easily seen that Definition 5 can be rephrased to:

Definition 6. A contract $C(\varphi)$ with vocabulary V *permits* a query ψ iff the intersection of the set of runs satisfying φ with the set of runs satisfying ψ, contains a projection class w.r.t. V.

This alternative definition is more suited for the design of an algorithm as we could think of computing the intersection of the two sets, and then checking the presence of a projection class in the intersection. The first part of this algorithm is very similar to the non-empty intersection problem, extensively studied in other domains (e.g. model checking [25]). To solve this problem the theo-

retical tool of choice are Büchi automata (BA), which have a deeply exploited connection with LTL [28]. Using BAs, we design a new algorithm that solves the problem with a single step: scanning the intersection of the two sets of runs and, simultaneusly, checking that it includes a complete projection class.

6.2.1 Büchi automata

Formally, a Büchi automaton is a tuple $\{Q, I, \delta, F\}$, where Q is the set of states, I the initial states and F the final ones. The transition relation is $\delta \subseteq Q \times \Sigma \times Q$, where Σ is the set of disjunction-free propositional formula over $\mathcal{V}$ (e.g. Figure 1).

In order to define the semantics we introduce the concept of *path* as an infinite sequence of states $\sigma = \sigma_1, \sigma_2, ...$, linked by transitions in δ (i.e. $\forall i(\exists\lambda(\langle\sigma_i, \lambda, \sigma_{i+1}\rangle \in \delta))$). A path is called *lasso path* if it is formed by a finite *prefix* leading to a final state k (called *knot*) and by a *cycle*, iterated infinitely many times, leading back to k. Intuitively, a run ρ *satisfies* a path σ if at any instant i the truth assignment $\rho(i)$ satisfies the formula labeling the transition from σ_i to σ_{i+1}. Following the result in [28], the accepting condition can be stated as: a BA $\mathcal{A}$ accepts a run ρ, if ρ satisfies a lasso path of $\mathcal{A}$. We call $BA(\varphi)$ any BA that accepts all and only the runs satisfying an LTL formula φ. We assume that the transition labels of $BA(\varphi)$ contain only variables in φ.

Consider now the problem of verifying that a contract permits a property. They are both specified as LTL formulae, but our approach will handle them in form of BAs. As in §3.1, we formalize this idea by introducing the concept of *simultaneous lasso path*.

Definition 7. Given a contract defined by φ (referred to as $C(\varphi)$) and a query ψ. Let $\mathcal{A}(\varphi) = \{Q_\mathcal{A}, I_\mathcal{A}, \delta_\mathcal{A}, F_\mathcal{A}\}$ be a BA for $C(\varphi)$ and $\mathcal{B}(\psi) = \{Q_\mathcal{B}, I_\mathcal{B}, \delta_\mathcal{B}, F_\mathcal{B}\}$ a BA for ψ, we define a *simultaneous lasso path* an infinite sequence of pairs $\langle s_i, q_i\rangle_{i\geq 0}$ s.t.:

1. $\langle s_i\rangle_{i\geq 0}$ form a lasso path for $\mathcal{A}$
2. $\langle q_i\rangle_{i\geq 0}$ form a lasso path for $\mathcal{B}$
3. for every i, there exist θ_i a formula s.t. $\langle s_i, \theta_i, s_{i+1}\rangle \in \delta_\mathcal{A}$ and τ_i s.t. $\langle q_i, \tau_i, q_{i+1}\rangle \in \delta_\mathcal{B}$, and we have that $\theta_i \wedge \tau_i$ is satisfiable (i.e. they are not conflicting), and τ_i contains only variables in $C(\varphi)$. We say that θ_i and τ_i are *compatible*.

The following theorem, proved in [6], formalizes the intuition.

THEOREM 3. *Let $C(\varphi)$ be a contract with variables in V and ψ a query. There exists a simultaneous-lassopath for $C(\varphi)$ and ψ iff $C(\varphi)$ permits ψ.*

As explained in §3.1, this theorem implies that the permission problem is decidable.

6.2.2 Algorithm

Our verification algorithm looks for a simultaneous lasso path in a way inspired by the nested depth first search technique used in model checking [25], which despite solving a different problem also relies on finding lasso paths.

Recalling that a lasso path is formed by a prefix and a cycle we now define similar concepts for simultaneous lasso paths. We call *knot* any pair $\langle s_i, q_i\rangle$ s.t. it appears infinitely many times in the simultaneous lasso path and q_i is a final state for the query BA. We call prefix w.r.t. a knot k, the portion of the simultaneous lasso path from the beginning to the first appearance of k. In order to define the cycle of a simultaneous lasso path, we have to consider the fact that the sequence $\langle s_i\rangle_{i\geq 0}$ has to be a lasso path for the contract BA. This means that we need to force the existence of a contract final state in the contract BA portion of the simultaneous lasso path. More formally, let i be the instant of the first appearance of a knot k and let $j > i$ be the instant of the first appearance of a pair $\langle s_i, q_i\rangle$ where s_i is a final state for the contract BA, we call *cycle* w.r.t. a knot k, the portion of the simultaneous lasso path between i and the first subsequent appearance of k after j.

Algorithm 1: Verifying that $C(\varphi)$ permits property ψ

```
   Input: A = {Q_A, i_A, δ_A, F_A} is the service BA,
   B = {Q_B, i_B, δ_B, F_B} is the query BA,
   (w.l.o.g. they have a single initial state)
   ψ, an LTL query property
   Output: returns true if S(φ) permits ψ
1  begin
2      visited = ∅
3      found = false
4      visit(⟨i_A, i_B⟩)
5      return found
6  end
7  procedure visit(⟨s, q⟩)
8  begin
9      if ⟨s, q⟩ ∈ visited then return
10     visited ⇐ visited ∪{⟨s, q⟩}
11     if q ∈ F_B then
12         visited2 ⇐ ∅
13         cycle_search(⟨s, q⟩, ⟨s, q⟩, false)
14         if found then return
15     foreach ⟨s, λ_A, s'⟩ ∈ δ_A do
16         foreach ⟨q, λ_B, q'⟩ ∈ δ_B s.t. compatible(λ_A, λ_B) do
17             visit(⟨s', q'⟩)
18             if found then return
19 end
20 procedure cycle_search(⟨s, q⟩, start, foundFinal)
21 begin
22     foundFinal2 ⇐ foundFinal ∨(s ∈ F_A)
23     if ⟨s, q⟩ ∈ visited2 then return
24     visited2 ⇐ visited2 ∪{⟨s, q⟩}
25     foreach ⟨s, λ_A, s'⟩ ∈ δ_A do
26         foreach ⟨q, λ_B, q'⟩ ∈ δ_B s.t. compatible(λ_A, λ_B) do
27             if ⟨s', q'⟩ = start then
28                 found ⇐ foundFinal2
29             cycle_search(⟨s', q'⟩, start, foundFinal2)
30             if found then return
31     visited2 ⇐ visited2 \{⟨s, q⟩}
32 end
```

In order to verify the existence of a prefix and a cycle, we consider the initial pair, then we consider all possible prefixes using a recursive depth first search: for every pair we inspect both input BAs (contract and query), build all possible successor pairs (i.e. those satisfying condition 3 of Def. 7) and recurse on each of those.

Since we do not know in advance the length of the prefix, at every step, if the considered pair $s = \langle s_i, q_i\rangle$ is a potential knot (i.e. q_i is a final state for the query BA), we verify if there exists a cycle containing s. To do so, we start a nested search in s which looks for a path that leads back to s and that also contains a pair $\langle s_i, q_i\rangle$ in which s_i is a contract final state. A straightforward way to look for such cycles is to perform a backtracking depth first search that keeps track of paths containing a pair with a contract final state and returns true iff one of these paths lead back to s. Algorithm 1 implements the strategy we just outlined. Lines 1 through 6 initialize the sets of visited pairs for the depth first visit, and the global variable that will contain true iff a simultaneous lasso path is found. It then invokes the procedure `visit` which recursively generates all prefixes.

Lines 9 and 10 verify that we are visiting every pair once. Lines 11 through 14 test if the current pair is a potential knot (q_i is a final state). If this is the case, the data structure $visited2$ is initialized and the procedure `cycle_search` is invoked. If it returns successfully, the procedure terminates. Lines 15-18 generate all possible successor pairs by inspecting the transitions in the contract BA

Table 1: LTL precedence pattern (s precedes p), from [9]

Global	$\mathbf{F}p \rightarrow (\neg p\mathbf{U}(s \vee \neg p))$
Before r	$\mathbf{F}r \rightarrow (\neg p\mathbf{U}(s \vee r))$
After q	$\mathbf{G}(\neg q) \vee \mathbf{F}(q \wedge (\neg p\mathbf{U}(s \vee \mathbf{G}(\neg p))))$
Between q and r	$\mathbf{G}((q \wedge \mathbf{F}r) \rightarrow (\neg p\mathbf{U}(s \vee r)))$

(line 15) and in the query BA (line 16). A recursive call to `visit` is then issued for each such pair. The predicate $compatible$ at line 16 (and later at line 32) implements condition 3 of Definition 7.

The procedure `cycle_search` accesses the $visited2$ variable which stores the path it is currently exploring. The variable $found\text{-}Final2$ is true if the current path contains a pair with a contract final state. Lines 23 and 24, avoid creating paths that contain cycles.Lines 25 and 26, generate all possible successor pairs. Line 27 and 28, update the global variable $found$ if a cycle is present which contains a pair with a contract final state. If this is not the case, line 29 and 30 recursively continue to extend the current path.Line 31 backtracks to the previous explored path by removing the current pair from the $visited2$ variable. In [6], we prove the result:

THEOREM 4. *Algorithm 1 returns true iff $C(\varphi)$ permits ψ.*

Note that the procedure $cycle\_search$ can exploit a simple memoization scheme which stores for every visited pair a boolean variable, which is true if that pair can eventually lead to the original node. In this way, we can code the whole procedure as a depth first visit, never visiting any pair more than once.

6.3 Simplified contract automaton

In this section we formalize the intuition of simplified contract automata, given in the context of our bisimulation index (§5). We say that a BA $\mathcal{A}$ is *equivalent* to a BA $\mathcal{B}$ w.r.t. a query BA $\mathcal{Q}$ iff there exists a simultaneous lasso path for $\mathcal{A}$ and $\mathcal{Q}$ iff there exists one for $\mathcal{B}$ and $\mathcal{Q}$. We consider a query BA $\mathcal{Q}$, and let $L_{\mathcal{Q}}$ be the set of literals that appear in labels of $\mathcal{Q}$. $L^n_{\mathcal{Q}}$ is the set that contains the negation of every literal in $L_{\mathcal{Q}}$.

Definition 8. We call the *relevant* BA $\mathcal{A}^r$ of a service BA $\mathcal{A}$ w.r.t. a query BA $\mathcal{Q}$, a BA that has the same states as $\mathcal{A}$ but that for every transition $\langle s, \chi, s' \rangle \in \delta_{\mathcal{A}}$ has $\langle s, \chi', s' \rangle \in \delta_{\mathcal{A}^r}$ s.t. χ' is a conjunction of all the literals in χ that are in $L^n_{\mathcal{Q}}$.

The following result formalizes Theorem 2 and is proved in [6]:

THEOREM 5. *Given a contract BA $\mathcal{A}$ and a query BA $\mathcal{B}$, there exists a simultaneous lasso path for $\mathcal{A}$ and $\mathcal{B}$ iff there exists one for its relevant BA $\mathcal{A}^r$ and $\mathcal{B}$.*

As with the minimization of traditional finite state automata [15], we use the bisimulation concept to reduce the number of states.

Definition 9. Let a and b be two nodes of a BA $\mathcal{A}$. We say that $a \sim b$ (read *bisimulates*) iff: (1) if $a \in F_{\mathcal{A}}$ then $b \in F_{\mathcal{A}}$ and vice versa, and (2) for every edge $\langle a, \lambda, a' \rangle \in \delta_{\mathcal{A}}$ we have $\langle b, \lambda, b' \rangle \in \delta_{\mathcal{A}}$ with $a' = b'$ or $a' \sim b'$, and vice versa.

We create equivalence classes of BA states (called $B(\mathcal{A})$) w.r.t. bisimulation, and we use them as the states of a new BA.

Definition 10. Given a BA $\mathcal{A}$, we call $\mathcal{A}_s$ (its *simplification*) a BA s.t. (1) $Q_{\mathcal{A}_s} = B(\mathcal{A})$, (2) $I_{\mathcal{A}_s}$ is any $s \in B(\mathcal{A})$ that contains a state in $I_{\mathcal{A}}$, (3) $F_{\mathcal{A}_s}$ is any $s \in B(\mathcal{A})$ that contains only states in $F_{\mathcal{A}}$, (4) let $C(a)$ be the bisimilarity equivalence class for the state a, for every $\langle a, \lambda, a' \rangle \in \delta_{\mathcal{A}}$ we have $\langle C(a), \lambda, C(a') \rangle \in \delta_{\mathcal{A}_s}$.

The simplification of a BA is equivalent to the original one in the following sense, proved in [6]:

THEOREM 6. *For every path $\sigma = \sigma_1, \sigma_2, ...$ for a BA $\mathcal{A}$ there exists a path $\sigma^s = \sigma^s_1, \sigma^s_2, ...$ for $\mathcal{A}_s$ and vice versa, where for all i, σ^s_i is the bisimilarity class of σ_i.*

Using Theorems 5 and 6, we prove in [6] the main result that enables our bisimulation index.

THEOREM 7. *Given a service BA $\mathcal{A}$ and a query BA $\mathcal{B}$, there exists a simultaneous lasso path for $\mathcal{A}$ and $\mathcal{B}$ iff there exists one for the simplification of its relevant BA $(\mathcal{A}^r)_s$ and $\mathcal{B}$.*

7. EXPERIMENTAL EVALUATION

7.1 Prototype Architecture

Our prototype is written in Java and consists of four independent modules. The first is the data generator and produces databases of LTL formulae as described in §7.2. It also converts the formulae to BAs using the freely available library LTL2BA [12]. Its output is a text file. The second module generates the prefiltering index from a text file containing contract specifications. The third precomputes the simplified BAs from a text file containing contract specifications. The fourth module is the runtime module. It uses the prefiltering index and the simplified BAs in order to evaluate queries. It takes as input a query workload text file and outputs statistics regarding their evaluation.

We executed all code on an AMD Phenom 2.2 GHz with 4Gb of RAM and we used Sun 32bit JVM 1.6.0.10. The code does not utilize multithreading in order to isolate the technique performance from system level effects.

7.2 Contract data generation

Given the novelty of our setting, it was impossible for us to find real databases of contract specifications. Towards a realistic generation, we adopted a sophisticated generation method based on real-life usage patterns of temporal properties ([9]). The work in [9] surveys over 500 real-life specifications of temporal properties to be formally verified on finite state systems (theoretically equivalent to our contracts) and extracts recurrently appearing patterns that cover over 92% of the surveyed cases, along with their occurring frequencies. While the specifications of these systems come from different application domains (e.g. communication protocols, GUIs, distributed object systems, operating systems, databases), we contend that the sort of properties that form these specifications are very similar to the ones found in service contracts, as they are effectively describing the allowed temporal sequences of interactions between multiple actors. In [9], patterns are classified along two dimensions: required behavior and scope (i.e. temporal interval in which the behavior has to show itself). The possible scopes include *global* (the whole timeline), *before* (up to certain event), *after* (after a given event), *between* (any part of the timeline between specified event p and event q). The possible behaviors are:

Absence An event does not occur in the scope.

Existence A given event must occur within a scope. A variation can force the occurrence number to k.

Universality A given event occurs throughout the scope.

Precedence An event p must always be preceded by an event q. A variation can consider sets of events instead of p and q.

Response An event p must always be followed by an event q. A variation can consider sets of events instead of p and q.

As an example, we show in Table 1 (from [9]) the LTL expression of the precedence behavior for the cited scopes.

Contracts and queries are generated as conjunctions of clauses expressed as LTL properties. Our generator randomly generates LTL properties using the distribution reported in [9]; at each generation it substitutes the variable placeholders (e.g. p, q) with variables from the common vocabulary. Given a parameter n, it generates a contract or query specification formed by the conjunction of n properties thus generated. We refer to [6] for detailed examples of our generated contracts and queries.

Table 2 shows the statistics of our generated datasets (i.e. contract databases and query workloads). Along with the name of the dataset we report its size (i.e. number of contracts or queries) and the number of LTL properties that form each contract or query. Since we found that the actual complexity of the LTL formulae is better characterized by the statistics of their associated BAs, we

Table 2: Datasets statistics

Dataset name	size	#LTL patterns	#states avg	#states stddev	#transitions avg	#transitions stddev
Simple contracts	3000	5	31.00	34.73	628.71	1253.37
Medium contracts	1000	6	41.82	43.23	964.69	1628.66
Complex contracts	1000	7	50.85	47.5	1291.63	1904.82
Simple queries	100	1	2.31	1.41	5.2	5.4
Medium queries	100	2	5.44	4.81	23.86	33.18
Complex queries	100	3	9.6	11.11	92.84	203.42

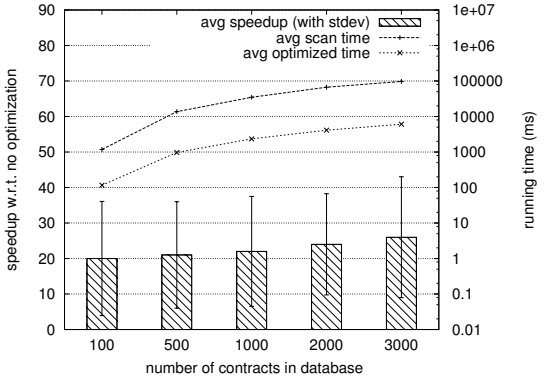


Figure 5: Average speedup and running times (unoptimized and optimized) w.r.t. database sizes (small contracts, average of all query complexities)

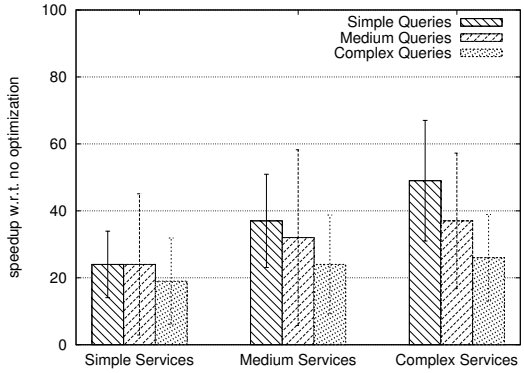


Figure 6: Average speedup w.r.t. contracts and query complexities (database size = 1000 contracts)

also report the average number of states of the BAs in the specification and its standard deviation along with the average number of transitions with its standard deviation.

7.3 Experiments

We measure the running times of query evaluation for the unoptimized system (§3) and for the system using both the prefiltering and the bisimulation techniques. The query evaluation time for the unoptimized system is the sum of the conversion time of the query from LTL to BA plus the evaluation on the contract database stored as set of contract BAs. The query evaluation time for the optimized system include the query conversion time plus the time used to evaluate the query using our precomputed data (i.e. prefilter index, simplified BAs) in addition to the contract BAs database.

A first batch of measurements aims to prove scalability w.r.t. the number of contracts in the database. We evaluate all queries (all complexity levels) on databases of sizes 100 to 3000 of simple contracts (subsets of the Simple Contract database in Table 2). For every contract database size, we record the average query evaluation time for the unoptimized and optimized systems. To some readers, these databases might seem too small for practical use. Recall, however, that a complete implementation would use relational attributes (e.g. travel date) to select the contracts that have to be checked for permission from a potentially much larger database.

A second batch of measurements aims at scalability w.r.t. complexities of both queries and contract specification. We evaluate individually every query database against contract databases of size 1000 (simple, medium, complex). For every combination of queries and contract complexities we record the query evaluation time for the unoptimized and optimized systems.

7.4 Results

First batch - Scaling w.r.t. database size. In Figure 5 the two lines at the top of the graph represent the average running time in the case of the unoptimized approach, called *scan time*, and the one of the optimized system, called *optimized time*. The times scale is on the right y axis and it is in milliseconds and logscale. The bar chart on the same picture shows the average speedup (along with standard deviation bars) achieved by the optimized system compared to the unoptimized one (the speedup scale is on the left y axis). Notice that the running times for both evaluations scale nearly linearly w.r.t. the size of the database. The unoptimized time ranges from 2sec for a small 100 contracts database to 100sec in the case of a 3000 contracts database. These times would limit the usage of our approach in interactive systems. Our optimization techniques, however, achieve an average speedup of at least 20 (scaling nicely for larger databases), reducing the average running time in the case of the 3000 contracts database to few seconds instead of minutes. Note that the speedups increase together with the size of the database, a common effect of indexing schemes. Finally, we rarely have speedups less than 10.

Second batch - Scaling w.r.t. complexity. Figure 6 reports the results for the scalabitility w.r.t. query and contract complexity. The bars represent the average speedups (along with standard deviation bars) of the optimized system compared to the unoptimized one w.r.t. complexities of both contracts in the database and queries. We explain the reduced speedup of complex queries noting that, using more variables, they cannot take advantage of the most simplified contract BAs. The bisimulation technique is also responsible for the increasing speedup w.r.t. complexities of the contracts. Usually complex contracts cite more variables and with the bisimulation technique the algorithm can ignore the behavior of all variables that are not used in the query.

Index building and size. For our optimized evaluation, each contract database required the precomputation of two data structures. The computation of the prefiltering index (§4.2) lasted less than 25 mins for our largest database of 3000 contracts. The average insertion time was around 500ms. The size is also minimal as our largest prefilter index (3000 contracts) was $\sim$10Mb.

The computation of the simplified BAs is more expensive as we compute all possible projections. For our 3000 contract database the average insertion time of a contract is 42secs, and the total computation of our largest datasets took 11 hours on our workstation using three cores. Since the workload is completely parallel (each contract is simplified independently), scaling the number of contracts can be tackled by adding resources. Moreover, to address increase in contract complexity, we can compute only the projections with few literals, while still obtaining significant benefits. As motivated at the end of §5.2, these projections would be used by small queries, which benefit the most from this technique. The size of the simplified BAs data is on average around 80% of the database original size and in absolute terms quite contained: our largest database of 3000 contracts had a combined size (database plus simplified BAs) of 112Mb.

8. RELATED WORK

To the best of our knowledge, this is the first work that deals with the discovery of contracts based on their temporal behavior. Previous works in electronic commerce and enterprise computing have dealt with the problem of managing contracts. The field of e-contracting [2] deals with the automation of contracting practices and it is mainly tailored for the B2B case. The direction that is most relevant to the present work is the verification and monitoring of real world contracts. Contracts are usually represented using rules written in some form of temporal and/or deontic logic (i.e. stating permissions, obligations and prohibitions), [16], [18]. Both [16],[18] perform verification on contracts specified with deontic constraints. These works focus on the internal consistency of contracts, leading to the need of more expressive specification language. Queries on a *database* of contracts are never considered and we are not aware of any contract indexing scheme. Moreover, in [16] the higher order logic used to represent contracts is internally translated to Büchi Automata, opening the way to the application of our indexing techniques also for such higher order formalisms. To the best of our knowledge, our work is the first dealing with indexing of databases of Büchi Automata.

Web Service search exhibits many similarities with our problem. Web Services are complex pieces of software that expose their functionality on the web. Many works have addressed this problem. Some of them do not consider the temporal behavior, focusing on search in their annotated natural language description ([8]), hence are not applicable to our context. In [13], a method based on graph matching is proposed in order to implement behavioral similarity search of web services. This system returns approximated results, expecting a domain-knowledgeable user to read the actual specification of the returned contracts. Our system, on the other hand, implements an exact query evaluation algorithm, which avoid this. In [21], web services are modeled as directed graphs with nodes representing either actions or messages. An exact evaluation algorithm is proposed to query for graph patterns in service specifications. We contend that such a system would not be suited for our context in which both contracts and queries are inherently declarative specifications and not directly related to a procedural implementation.

The problem of querying behavioral objects is present also in the area of business process management. In [3], business processes are represented as directed graphs encoding the BPEL specification. An exact query evaluation algorithm is presented that allows to query the *structure* of the specification graphs using graph queries. The framework is extended in [7], in order to consider some behavioral semantics. The premise of this line of work differs from ours in that business processes are handled as *procedural* implementations (workflows), while in our work we manage contracts in the form of *declarative* clauses. Asking for a full workflow registration by the contract provider would significantly hinder registration into the contract database, as contracts specify only *some* key properties of the workflow. Moreover, a workflow query will likely miss all contracts whose workflows do not match its structure yet are semantically equivalent to one that does.

Finally, many techniques we used are inspired by the work done in model checking [25]. However, our permission problem is a novel variation on the standard verification of satisfiability of LTL formulae, and solving it required adapting and combining these techniques in a novel way.

9. CONCLUSIONS

We present a broker that enables providers to register service contracts and consumers to query them, in both cases based on the contracts' temporal behavior. Our novel semantics for permission of a query by a contract takes into account the fact that contracts may not mention some of the events of interest to the query. Permission does not reduce to any standard temporal decision problem, and requires an original solution. We establish the complexity of the permission problem (PSPACE-complete), and design an algorithm for checking it. We show that the theoretical worst case is not an impediment to scalable implementation, presenting two distinct and complementary indexing techniques for querying large collections of service contracts. We evaluate experimentally our implementation, showing that it scales well with both the number and the complexity of the contracts.

10. REFERENCES

[1] U. Airlines. Contract of carriage. http://www.united.com/page/article/0,6867,2671,00.html.

[2] S. Angelov and P. Grefen. The business case for b2b e-contracting. In *Proc. of the 6th Int. Conf. on Electronic Commerce*, 2004.

[3] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *VLDB '06*, pages 343–354. VLDB Endowment, 2006.

[4] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*. 1996.

[5] M. Brambilla, A. Deutsch, L. Sui, and V. Vianu. The role of visual tools in a web application design and verification framework : A visual notation for ltl formulae. *Int. Conf. on Web Engineering*, 2005.

[6] E. Damaggio, A. Deutsch, and D. Zhou. Querying contract databases based on temporal behavior. Technical report, 2010.

[7] D. Deutch and T. Milo. Querying structural and behavioral properties of business processes. In *Database Programming Languages*. 2007.

[8] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. In *VLDB*, 2004.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Workshop on Formal Methods in Software Practice*, 1998.

[10] K. Fisler and M. Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In *Formal Methods in Computer-Aided Design*, 1998.

[11] E. Fredkin. Trie memory. *Commun. ACM*, 1960.

[12] P. Gastin and D. Oddoux. Fast ltl to büchi automata translation. In *CAV*, 2001.

[13] D. Grigori, J. C. Corrales, and M. Bouzeghoub. Behavioral matchmaking for service retrieval. In *Int. Conf. on Web Services*, '06.

[14] G. J. Holzmann. The model checker spin. *IEEE Trans. Soft. Eng.*, 1997.

[15] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[16] M. Kyas, C. Prisacariu, and G. Schneider. Runtime monitoring of electronic contracts. In *Automated Technology for Verification and Analysis*, 2008.

[17] M. R. Lowry. Software construction and analysis tools for future space missions. In *European Conf. on Theory and Practice of Software*, 2002.

[18] Z. Milosevic and R. G. Dromey. On expressing and monitoring behaviour in contracts. In *EDOC*, '02.

[19] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 1987.

[20] A. Pnueli. The temporal logic of programs. In *FOCS*, 1977.

[21] Z. Shen and J. Su. Web service discovery based on behavior signatures. In *Int. Conf. on Services Computing*, 2005.

[22] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 1985.

[23] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *STOC '73: ACM Symp. on Theory of Computing*.

[24] H. Strasberg. Travel terminal. http://www.travelterminal.com.

[25] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, 1986.

[26] Wikipedia. List of passenger airlines. http://en.wikipedia.org/wiki/List_of_passenger_airlines.

[27] Wikipedia. Travel class. http://en.wikipedia.org/wiki/Travel_class.

[28] P. Wolper, M. Y. Vardi, and P. A. Sistla. Reasoning about infinite computation paths. In *FOCS*, 1983.