Relational Transducers for Electronic Commerce

Serge Abiteboul*
I.N.R.I.A.-Rocquencourt
Serge.Abiteboul@inria.fr

Victor Vianu*
U.C. San Diego
vianu@cs.ucsd.edu

Brad Fordham
Oracle Corporation
bfordham@us.oracle.com

Yelena Yesha*
UMBC/CESDIS
yeyesha@cs.umbc.edu

Abstract

Electronic commerce is emerging as one of the major Web-supported applications requiring database support. We introduce and study high-level declarative specifications of business models, using an approach in the spirit of active databases. More precisely, business models are specified as *relational transducers* that map sequences of input relations into sequences of output relations. The semantically meaningful trace of an input-output exchange is kept as a sequence of *log* relations. We consider problems motivated by electronic commerce applications, such as log validation, verifying temporal properties of transducers, and comparing two relational transducers. Positive results are obtained for a restricted class of relational transducers called *Spocus transducers* (for semi-positive outputs and cumulative state). We argue that despite the restrictions, these capture a wide range of practically significant business models.

1 Introduction

Electronic commerce is emerging as a major Web-supported application. In a nutshell, electronic commerce supports business transactions between multiple parties via the network. This activity has many aspects, including security, authentication, electronic payment, and designing business models [YA96]. Electronic commerce also requires database support, since it often involves handling large amounts of data (e.g. product catalogs, yellow pages, etc.) and must provide transactions, concurrency control, distribution and recovery. In this paper, we argue that a database approach can provide the backbone for a wide range of electronic commerce applications. Beyond a supporting role, databases can provide high-level specification of the semantics of electronic commerce applications, in the form of declarative specifications of business models. Since business models specify a protocol of exchanges among partners to a transaction, their semantics is primarily behavioral. Their specifications therefore have a strong dynamic component, reminiscent of active databases and transactional workflows. However, the electronic commerce context raises new, specific issues, which are the focus of the present paper.

Business models are formalized as follows. The state of an application is described by a relational database. The interaction from the outside world is captured by a sequence of input relations. The application responds by a sequence of output relations. Thus, the model can be viewed as a machine that translates an input sequence of relations into an output sequence of relations. We call such a machine a *relational transducer*.

Like transducers in language theory, relational transducers are specified by a state transition function and an output function. In principle, this can be done in any programming language. However, in the electronic commerce context it is particularly important that the specification of a business model be easy to understand, for various reasons: the participants in the exchange must understand each other's business models, the business models themselves can be subject to negotiation, and business model specifications may carry contractual values. Therefore, a high-level, declarative specification of business models is particularly desirable. This motivated us to focus on simple rule-based specifications of relational transducers.

The semantics of a relational transducer is the mapping it induces from input sequences to output sequences. However, there is an important variation, motivated by electronic commerce applications. In many cases, only some of the inputs and outputs are semantically significant, while others represent syntactic sugaring that render the interface more user-friendly. For example, payment and delivery of a product might be considered significant, whereas inquiries about prices or reminders of pending bills might not. To capture this distinction we use the notion of a *log*, which is the restriction of an input-output sequence to designated relations. In many circumstances we consider the semantics of relational transducers relative to specified logs.

The problems we study relate to the design and verification of business models specified as relational transducers. A first class of problems relates to individual transducers, and includes the following.

*Work performed in part while S. Abiteboul was visiting Stanford University and while V.Vianu was visiting I.N.R.I.A. S. Abiteboul supported in part by CESDIS. V. Vianu supported in part by the National Science Foundation under grant number IRI-9221268. Y. Yesha supported in part by the IBM Center for Advanced Studies, Toronto, Canada.

- *Log validity*: testing whether a given log sequence can actually be generated by some input sequence. This problem arises for instance when, for efficiency and convenience, the relational transducer of a supplier is allowed to run on a customer's site. The trace provided by the log allows the supplier to validate the transaction carried out by the customer.
- *Goal reachability*: most business models are geared towards achieving a certain goal, such as delivering a product under certain conditions. Goal reachability checks whether a set of goals can be reached by some run of the transducer.
- *Temporal properties*: verification of desired temporal properties of the business model, such as "*No product can be delivered before payment is received.*"

We also consider problems involving more than one relational transducer. The most important is *containment*, i.e., testing whether every valid log of one transducer is also valid for another. The main motivation for considering this question is customization of business models. Current electronic business models tend to impose on users unnecessary constraints and limitations. It would be desirable to let the users customize the basic business model for their convenience or to conform to their own regulations. Then it is necessary to verify that the new business model still conforms to the original semantics, i.e. the valid logs of the customized model remain valid in the original model. A weaker criterion than containment is *compatibility*. Suppose two business partners have their own procedures for conducting business, codified in their respective business models. These models may well be contradictory – e.g., a customer may require delivery before payment, whereas a supplier may require payment before delivery. Compatibility verifies that there *exists* a run which achieves some desired goals while satisfying both business models.

Obviously, problems such as the above are undecidable for unrestricted relational transducers (say, with state and output functions defined by first-order means). One of the main objectives of this paper is to propose a restricted class of transducers which satisfies the following requirements: it is specified in a simple, declarative fashion; it is rich enough to specify a wide range of practically significant business models; and, questions such as the ones above can be effectively answered. We propose such a restricted model, called *S*emi-*po*sitive *cu*mulative *s*tate (*Spocus*) transducers. In a Spocus transducer, the state simply accumulates all inputs received. Outputs are defined from the state, current input, and database by a non-recursive, semi-positive set of datalog$^{\neg}_{\neq}$ rules. We argue that this simple model still captures a practically significant set of business models. In particular, we prove that it can enforce a useful class of temporal properties on runs. On the other hand, we show that many of the questions above are decidable for Spocus transducers. For most decision procedures, the complexity is NEXPTIME (but NP if the number of variables is fixed). While this may appear high, one should note that the complexity is in line with other classical decision problems on queries, such as containment of conjunctive queries. Also, most of our questions involve *static* analysis of transducers. We also show undecidability if we remove some of the restrictions of Spocus transducers. These results suggest that Spocus transducers achieve an appealing balance between expressiveness and tractability.

Related work Our work is motivated by electronic commerce (e.g., see [YA96]). We use the framework of relational databases [AHV95, Ull89]. Our relational transducers can be viewed as active databases with immediate triggering [WC95, PV97], although the questions we consider are quite different. The logic background is relational calculus [AHV95] for the static aspect and temporal logic [Eme91] for the temporal aspect. Since in some sense we deal with interacting processes, our work could be viewed in the more general context algebra and calculi for concurrent processes, e.g., [Mil91]. Also related to business models are Petri nets (e.g., [Rei83]). Our notion of transducer equivalence based on a log is in the spirit of the observational equivalence (e.g. [Par81, Mil80]). Business models are also related to workflow management (e.g. [work93].) A workflow typically involves performing distributed tasks in an enterprise. In this context, our approach in that context can be viewed as data-centric in the sense that it focuses on the interaction of a relational store with the external world.

One may question the need to introduce yet a new model rather than adopting an existing one. There are many practical reasons to build electronic commerce applications around a relational database. Furthermore, we believe that the simplicity of a declarative, logic-based approach is a clear advantage in specifying business models. Finally, our approach naturally leads to a set-at-a-time treatment of inputs, for which database query optimization and parallel evaluation techniques can be used.

The development of electronic commerce applications using active databases is considered in [FAY97]. This work motivates and backs up some of the ideas in the present paper. In [FAY97], a prototype system in the spirit of active databases for specifying electronic commerce applications is described. This is based on Gurevich's evolving algebras [Gur94] and is build on top of a Postgres [SR86] database. Applications were implemented using the prototype, and used in a production environment. However, the specification language was too rich to allow for formal verification and sometimes lead to descriptions of business models not easy to understand by users. This was the prime motivation for the present work.

Organization Section 2 introduces business models and the main problems addressed in the paper by means of an informal discussion, then formally defines relational transducers. Section 3 presents the Spocus model and the decidability and undecidability results. The last section provides brief conclusions.

2 Relational Transducers

We begin with an informal discussion and examples, then formally define relational transducers.

2.1 Informal discussion

In the first example we consider a very simple business model where a customer orders a product, is billed for it, pays, and then takes delivery. More precisely, a company may decide to provide the following business model:

```
TRANSDUCER SHORT
%
schema
  database: price, available;
  input: order, pay;
  state: past-order, past-pay;
  output: sendbill, deliver;
  log: sendbill, pay, deliver;
state rules
  past-order(X) +:- order(X);
  past-pay(X,Y) +:- pay(X,Y);
output rules
  sendbill(X,Y) :- order(X), price(X,Y),
                   NOT past-pay(X,Y);
  deliver(X)    :- past-order(X), price(X,Y),
                   pay(X,Y), NOT past-pay(X,Y).
```

Such a program specifies a *relational transducer*. It consists of three parts: a schema specification (database, input, output, state, and log relations), a state transition program and an output program. In the example, the database relations are *available* and *price*. (Ignore *available* for the moment.) A customer interacts with the system by inserting tuples in two input relations, *order* and *pay*. The system responds by producing output relations *sendbill* and *deliver*. Imagine that the presence of tuples in these relations is followed by actual actions such as sending an email with the bill and physically delivering the product. The system keeps track of the history of the business transaction using the state relations, here *past-order* and *past-pay*. The state and output relations are defined, respectively, by a state program and an output program. When a new set of input facts arrives, the transducer reacts by executing simultaneously the state program and the output program. The rules in the example have the obvious semantics, and are fired in parallel. The "+" in the state rules indicate that the semantics is cumulative, so the state relations simply contain all previously input facts. The output is not cumulative. A *run* of a transducer consists of (i) a sequence of inputs, (ii) the sequence of outputs generated by the transducer in response to each of the inputs, and (iii) the restriction of the input-output sequence to the relations in the log. Note that a run is completely determined by the sequence of inputs. The input and output sequences of a run of SHORT are shown in Figure 1. (The prices of $Time$, $Newsweek$, $LeMonde$ are \$55, \$45 and \$350, respectively.)

As noted earlier, the last component of the schema consists of the *log* relations, which are a subset of the input and output relations[1]. These relations are the ones that are considered semantically significant, for various reasons: they may result in actions with important consequences, they may carry legal meaning, etc. The validity of a run of the transducer is defined by what happens to the relations in the log. The restriction of the input-output sequence of a run to the log relations is simply called the *log of the run*.

The above transducer describes a very simple business model. We will shortly see a more realistic one. While in general one might use arbitrarily complex state and output programs, we will advocate the use of simple programs in the style of the above, that are sufficient in many practical cases, are easy to understand, and for which some properties of interest can be statically verified. We next mention some of these properties.

[1] If no log is specified, it is assumed that that the log consists of all inputs and output relations.

Log checking The first problem is related to fraud detection. For convenience and efficiency, one might allow certain customers to conduct business with the supplier by running locally the supplier's business model. As a record, the supplier is provided with the log of the run. To detect possible fraud, the supplier should be able to verify that the log is *valid*, i.e. it is a log allowed by the supplier's model. More precisely, given a log, the supplier has to verify that there exists a sequence of inputs that generates the log. Obviously, the problem is trivial if all inputs are logged. However, using a partial log makes sense if the log is much smaller than the full run, since the point of running the transducer at the customer site is to reduce the amount of data exchanged over the network.

Minimizing the log Related to the above is the problem of minimizing a given log, that is, finding a minimum log that is sufficient to reconstruct the given log for all runs. For instance, it is easy to see that in the previous transducer, one can remove the relation *deliver* from the log without losing any information. Indeed, it is easy to reconstruct its occurrences in a run from the occurrences of *order*, *price* and *pay*, and the given program. In general, there is a trade-off between shorter log and ease of verification.

Goal reachability and progress Goal reachability asks if some goal can be achieved by some run of the transducer, possibly with some preconditions. For SHORT one can verify that it is possible to achieve the goal $deliver(x)$ as long as $\exists y\, price(x, y)$ holds in the database. In general however, the problem can be much more complicated.

The notion of *progress* is related to the same question. It is classically the case that customers get lost in the intricacies of business models. In a given state, a user interested in achieving some goal such as $deliver(pc8000)$ may wish to be told what is the next action (input) that will make the system progress towards the goal.

Checking temporal properties A question of a slightly different flavor is verifying temporal properties satisfied by *all* runs. For instance, the supplier may wish to verify that a product is never delivered before it has been paid. Using modal operators with the obvious semantics, this amounts to verifying the following temporal formula:

$$\forall x \forall y\ always[(deliver(x) \wedge price(x, y)) \rightarrow sometime_past(pay(x, y))]$$

Modifying and comparing relational transducers The SHORT program captures the basic semantics of a simple application but is clearly not very user friendly. For instance, if a user orders an unavailable product, no warning is output. The following program recasts SHORT in friendlier terms:

```
TRANSDUCER FRIENDLY
%
relations
  database: price, available;
  input: order, pay, pending-bills;
  state: past-order, past-pay;
```

input sequence	$order(Time)$ $order(Newsweek)$ $order(Hustler)$	$order(LeMonde)$ $pay(Time, 55)$ $pay(Newsweek, 48)$	$pay(Newsweek, 45)$
output sequence	$sendbill(Time, 55)$ $sendbill(Newsweek, 45)$	$deliver(Time)$ $sendbill(LeMonde, 350)$	$deliver(Newsweek)$

Figure 1: Input and output sequences of a run of SHORT

input sequence	$order(Time)$ $order(Newsweek)$ $order(Hustler)$	$order(LeMonde)$ $pay(Time, 55)$ $pay(Newsweek, 48)$	$pay(Newsweek, 45)$	*pending-bills*()
output sequence	$sendbill(Time, 55)$ $sendbill(Newsweek, 45)$ $unavailable(Hustler)$	$deliver(Time)$ $rejectpay(Newsweek)$ $sendbill(LeMonde, 350)$	$deliver(Newsweek)$	$rebill(LeMonde, 350)$

Figure 2: Input and output sequences of a run of FRIENDLY

```
  output: sendbill, deliver, unavailable,
          rejectpay, alreadypaid,  rebill;
  log: sendbill, pay, deliver;
state rules
  past-order(X) +:- order(X);
  past-pay(X,Y) +:- pay(X,Y);
output rules
  sendbill(X,Y) :- order(X), price(X,Y),
                   NOT past-pay(X,Y);
  deliver(X)    :- past-order(X), price(X,Y),
                   pay(X,Y), NOT past-pay(X,Y);
  unavailable(X :- order(X), NOT available(X);
  rejectpay(X)  :- pay(X,Y), NOT past-order(X);
  rejectpay(X)  :- pay(X,Y), past-order(X),
                   NOT price(X,Y);
  alreadypaid(X :- pay(X,Y), past-pay(X,Y);
  rebill(X,Y)   :- pending-bills(), order(X),
                   price(X,Y),
                   NOT past-pay(X,Y).
```

One run of this program is shown in Figure 2.

The program FRIENDLY is obviously more customer friendly than SHORT. It issues warning messages when the product is unavailable, the payment is incorrect or the item has already been paid. It also answers requests for reminders of pending bills. One can easily verify that SHORT and FRIENDLY yield exactly the same set of valid logs. So, from a semantic viewpoint, they are interchangeable. Thus, the customer can be allowed to customize SHORT to FRIENDLY without violating the original model.

Customization raises the problems of containment and equivalence of relational transducers relative to a specified log. This is somewhat similar to observational equivalence in the style of [Mil91]. Note that *containment* of valid logs may well be acceptable as a criterion for soundness of customization: it guarantees that the valid logs of the customized transducer are still valid with respect to the original. To see an example, suppose a customer's internal regulations limit the use of this electronic model to purchases under 100K, or disallow buying some specific products from this particular supplier. It is easy to modify FRIENDLY to impose such constraints. The resulting set of valid logs is then strictly contained in the set of valid logs for SHORT. This remains acceptable to the supplier.

To conclude this section, we mention a class of important questions that are not addressed in the present paper. They are concerned with the *interaction* of transducers. The problem arises when each participant in an exchange has her own business model codified as a relational transducer. Then outputs of some transducers are fed as inputs to other transducers, possibly generating feedback loops. This raises questions such as the consistency of a system of transducers.

2.2 A formal model

We assume some familiarity with the relational model (e.g., see [AHV95]). Let $\mathbf{R}$ be a relational schema. A *sequence over* $\mathbf{R}$ is a finite sequence $I_1, ..., I_n$ where each I_i is a finite instance of $\mathbf{R}$.

A *transducer schema* is an expression

$$(\mathbf{in}, \mathbf{state}, \mathbf{out}, \mathbf{db}, \mathbf{log})$$

where each of the five components is a relational schema, the first four are pairwise disjoint and $\mathbf{log} \subseteq \mathbf{in} \cup \mathbf{out}$. Let $\mathbf{schema}$ be a transducer schema. A *relational transducer* over $\mathbf{schema}$ is a triple $(\mathbf{schema}, \sigma, \omega)$ where σ (ω) is a mapping of instances of $(\mathbf{in}, \mathbf{state}, \mathbf{db})$ to instances of, $\mathbf{state}$ ($\mathbf{out}$). The mapping σ is called the *state function* and ω the *output function*.

Given a sequence $I_1, ..., I_n$ over $\mathbf{in}$ (called an *input sequence*), and a database instance D of $\mathbf{db}$, the *run* of T *on* $I_1, ..., I_n$ and D is a state sequence $S_1, ..., S_n$, an output sequence $O_1, ..., O_n$, and a log sequence $L_1, ...L_n$ defined as follows: for each i in $[1..n]$,

1. $S_i = \sigma(I_i, S_{i-1}, D)$;
2. $O_i = \omega(I_i, S_{i-1}, D)$;
3. $L_i = (I_i \cup O_i)|_{\mathbf{log}}$;

where S_0 is empty. We denote by $state_T(D, I_1, ..., In)$, respectively $out_T(D, I_1, ..., In)$ and $log_T(D, I_1, ..., In)$, the state, output and log sequences of the run of T on $I_1, ..., I_n$ and D; T and D are omitted when they are understood.

Recall from the informal discussion that the role of the $\mathbf{in}$ relations is to describe the inputs from users of the system. The $\mathbf{db}$ relations represent a database used by the system (possibly very large and external). The $\mathbf{state}$ relations represent the information that the system remembers from its current run. The $\mathbf{out}$ relations capture the reactions of the system, and the $\mathbf{log}$ relations designate the semantically significant inputs and outputs. If $\mathbf{log} = \mathbf{in} \cup \mathbf{out}$ then we say the log is *full*; otherwise, it is *partial*.

Note that one could write programs where a copy of the current input is included in the output, so we could without loss of generality restrict the log to contain only output relations. Also note that **db** could be merged into the state relations. However, the distinctions become important when restricted classes of transducers are considered.

Let us now restate in terms of relational transducers some of the questions we will study, so far discussed informally. For a relational transducer T and a database D, we define the following problems:

log validity given a sequence $L_1, ..., L_n$ over **log**, is there an input sequence $I_1, ..., I_n$ such that $L_1, ..., L_n$ is the log on input $I_1, ..., I_n$? More formally, is the following true:

$$\exists I_1, ..., I_n (L_1, ..., L_n = log(I_1, ..., I_n))$$

goal reachability does a given sentence φ over **out** (a "goal") hold in the last output of *some* run of T on database D?

A variation of the question asks if a goal φ is reachable after a partial run $R_1, \ldots, R_m$. That is, is there a continuation $R_{m+1}, \ldots, R_n$ of the run such that φ is satisfied in the last output of the run $R_1, \ldots, R_n$?

containment and equivalence Let T and T' be transducers with the same log relations. Log containment asks if every valid log of T is also a valid log of T'. Equivalence asks if T and T' have the same set of valid logs:

$$\begin{array}{ll} (T \subseteq T') & \forall I_1, ..., I_n \exists J_1, ..., J_n \\ & (log(I_1, ..., I_n) = log(J_1, ..., J_n)) \\ (T \equiv T') & T \subseteq T' \text{ and } T' \subseteq T \end{array}$$

3 Spocus Transducers

In this section, we focus on a simple class of relational transducers, called Spocus transducers, which is appealing for two reasons. First, many of the questions discussed above become decidable in this setting. Second, as we shall argue, the language remains powerful enough to specify many business models of practical interest. We first define the Spocus transducers, then show the decidability of several important properties involving single transducers. We also consider the containment of transducers. Finally, we explore the use of transducers as acceptors, which allows specifying restrictions on valid input and output sequences.

3.1 Definition

Spocus transducers are relational transducers restricted as follows: the state relations simply accumulate the inputs; and output relations are defined by non-recursive, semi-positive datalog programs with inequality. (*Spocus* stands for *S*emi-*po*sitive *o*utput and *cu*mulative *s*tate.) Formally, we have:

Definition: Let $\mathbf{schema} = (\mathbf{in}, \mathbf{state}, \mathbf{out}, \mathbf{db}, \mathbf{log})$ be a transducer schema. A *Spocus transducer* is a relational transducer $(\mathbf{schema}, \sigma, \omega)$ such that:

1. $\mathbf{state} = \{past\text{-}R \mid R \in \mathbf{in}\}$, where $past\text{-}R$ has the same arity as R;
2. for each i,

$$\sigma(I_i, S_{i-1}, D)(past\text{-}R) = S_{i-1}(past\text{-}R) \cup I_i(R)$$

for every $R \in \mathbf{in}$; and

3. $\omega(I_i, S_{i-1}, D)$ is defined by a finite set of rules of the form $A_0 \leftarrow A_1, \ldots, A_n$ where:
   - A_0 is a positive literal $R(\vec{x})$ where $R \in \mathbf{out}$,
   - A_i is of the form $(\neg)R(\vec{x})$ or $x \neq y$ where $R \in \mathbf{in} \cup \mathbf{state} \cup \mathbf{db}$,
   - each variable in the rule occurs positively in the body of the rule.

The semantics of the program ω is the standard one for semi-positive datalog programs.

The transducers SHORT and FRIENDLY turn out to be Spocus transducers. We will show that, despite their simplicity and very limited use of state relations, Spocus transducers have significant control capabilities. To provide some intuition, we first illustrate this by considering Spocus transducers whose inputs and outputs are propositional, and which further output at most one proposition at a time. We call these *propositional transducers*. The set of output sequences generated by such a transducer T, denoted $Gen(T)$, can then be viewed as words over the finite alphabet of output propositions. For instance, the following Spocus transducer generates all prefixes of words in the language ab^*c.

```
input relations A, B, C;
state relations past-A, past-B, past-C;
output relations a, b, c;
state rules
  past-A +:- A; past-B +:- B; past-C +:- C;
output rules
  a :- A, NOT past-A;
  b :- B, past-A, NOT past-C, NOT C;
  c :- C, past-A, NOT past-C.
```

Note that we cannot control the input but only the output. For instance, we cannot prevent A to be input several times, but, using $past\text{-}A$, we can guarantee that a is produced at most once.

We can characterize precisely the languages generated by propositional transducers. They are the prefix-closed regular languages accepted by finite automata with no cycles except self loops. Intuitively, this is due to the inflationary nature of states in Spocus transducers: one can never return to a previous state. Clearly, the prefix closure of ab^*c is such a language, whereas the prefix closure of $(ab)^*$ is not.

Recall that $Gen(T)$ refers only to outputs; inputs remain unrestricted. We examine in Section 3.4 a mechanism for restricting inputs, that increases expressiveness dramatically.

3.2 Verifying a single Spocus transducer

We next return to several of the basic questions on transducers formulated in the previous section and show their decidability in the framework of Spocus transducers. In some cases, we show how slight strengthening of the Spocus model leads to undecidability. We also mention some open questions along the way. We begin with properties of single transducers (log validation, goal reachability, properties of runs). By way of preliminaries, we note that most of the

decidability results are shown by reduction to finite satisfiability of FO sentences with relational vocabulary, constants, and equality, of the form

$$\exists x_1 \ldots \exists x_k \forall y_1 \ldots \forall y_m \ \varphi(x_1, \ldots, x_k, y_1, \ldots, y_m)$$

where φ is quantifier-free. This is the well-known Bernays-Schönfinkel prefix class [BS28], which we denote $\exists^*\forall^*$FO. We use similar notation for prefix classes such as $\exists^*$FO and $\forall^*$FO, with the obvious meaning. The decidability of finite satisfiability of $\exists^*\forall^*$FO sentences was shown in [Ram30], and it was proven in [Lew80] that the problem is complete in NEXPTIME (but in NP for fixed number of universal quantifiers, see also [BGG97]).

Log validation The question is clearly trivial when the log contains all the inputs: just run the transducer on the given input sequence and database and verify that this log is indeed obtained. For transducers with a partial log, we have:

Theorem 3.1 Given a Spocus transducer T, a fixed database instance D and a log $\mathcal{L}$, it is decidable [2] in D^P (NP if the number of variables in T is fixed) whether $\mathcal{L}$ is valid, i.e., $\mathcal{L} = log_T(D, \mathcal{I})$ for some input sequence $\mathcal{I}$.

Proof: A log $\mathcal{L} = L_1 ... L_n$ is valid if there exists an input sequence $\mathcal{I} = I_1 ... I_n$ that generates it. We can view $\mathcal{I} = I_1 ... I_n$ as an instance over a relational schema obtained by replicating n times each input relation R, yielding $R_1 ... R_n$. The problem can then be reduced to the question of the satisfiability of an $\exists^*\forall^*$FO sentence over this (extended) relational schema. To state that $\mathcal{I}$ yields the log $\mathcal{L}$, we must state that the input relations in $\mathcal{I}$ recorded by the log have the values specified by $\mathcal{L}$, as well as the output relations determined by $\mathcal{I}$. Saying that a tuple belongs to an input (output) relation can be done by an $\exists^*$FO sentence. Saying that the only tuples in an input (output) relation are the ones in the log requires a $\forall^*$FO sentence (for output relations, this is possible due to the restricted form of rules defining the outputs in Spocus transducers, including the fact that each variable in a rule must occur positively). Overall, this can be written as an $\exists^*\forall^*$FO sentence, whence decidability in NEXPTIME. A slightly more careful analysis actually shows that the complexity is D^P. If the number of variables in T is fixed, the complexity is NP because a fixed number of universal quantifiers are needed. □

Note that a similar result holds if the database is not known, i.e., one can decide whether there *exists* a database over **db** for which the given log is valid.

We conclude this section with two remarks on the previous result:

(1) Spocus Restrictions: Log validation is quite expensive. Indeed, it remains NP-hard even if output relations are defined by conjunctive queries over the state relations only. (This is because view consistency is NP-hard for views defined by conjunctive queries [AD97].) Other problems however are simplified for such transducers. For example, reachability of a positive goal is decidable in PTIME. Restrictions of Spocus transducers, and their impact on the complexity of decision procedures considered here, need to be further investigated.

(2) Spocus Extensions: Spocus transducers were designed so that questions such as log validity are decidable. Some of the restrictions placed on Spocus rules could be slightly relaxed. For example, log validity (and other problems) remains decidable if states are defined by positive rules with no free variables in their body. If such variables are allowed in state rules (which amounts to allowing projection) log validity becomes undecidable. This can be shown by reduction of the implication problem for functional and inclusion dependencies (FDs and IncDs) which is undecidable [CV85, Mit83]. Given sets F, G of FDs and IncDs over some relation R, a transducer with such state rules can do the following: (i) on input R, store in the state relations R together with its projections involved in the IncDs in F and G (ii) at the next step, output *violation-F* is F is violated and *violation-G* is G is violated by the stored R; this can be checked by output rules using the stored projections. The log consists of *violation-F* and *violation-G*. Clearly, $F \not\models G$ iff the log $\langle \emptyset, \{violation\text{-}G\} \rangle$ is valid. We illustrate the construction for a binary relation R, $F = 1 \rightarrow 2$, and $G = R[1] \subseteq R[2]$ (in this case $F \not\models G$). We obtain the following transducer:

```
input relations R;
state relation past-R, R2;
state rules
  past-R(x,y) +:- R(x,y);
  R2(y)       +:- R(x,y);          % not Spocus
output rules
  violation-F  :- past-R(x,y), past-R(x,y'),
                  y <> y';
  violation-G  :- past-R(x,y), NOT R2(x).
```

Goal reachability Business models often aim at achieving a particular goal, such as delivering a product. Given such a model, a minimum sanity check is to make sure the model allows one to achieve the goal. We formalize this as follows. A *goal* γ is a sentence of the form $\exists \vec{x}(A_1 \wedge \ldots \wedge A_k)$ where each A_i is a positive or negative literal over an output relation, and each variable occurs in some positive literal. Let T be a relational transducer and γ a goal. Goal reachability asks if there is a run of T such that γ is satisfied by the last output. We can show:

Theorem 3.2 Given a Spocus transducer T and a goal γ, it is decidable in NEXPTIME (NP if the number of variables in T is fixed) if there exists a run of T whose last output satisfies γ.

Proof: The proof is in the same spirit as that of Theorem 3.1. It consists of two parts: First, find a bound k on the length of runs that need to be considered (the bound equals the number of positive literals in γ). Second, reduce the problem to the satisfiability of an $\exists^*\forall^*$FO sentence over a schema consisting of k copies of the input. If the number of variables in T is fixed, then the number of universal quantifiers is fixed, which yields the NP complexity. □

Note that, although limited to output relations, goals as above can also be used to make simple temporal statements about runs, which involve the entire history of inputs. Technically, this can be shown by including in the output the relevant part of the database, state and current input. Thus,

[2] D^P denotes the properties expressible as $\sigma \wedge \psi$ where σ is in NP and ψ is in CO-NP.

one can check temporal statements of the type "*deliver*(x) cannot be output unless *pay*(x,y) has been previously input, where *price*(x,y) is in the database". We next explore more formally such temporal statements and more general ones as well.

Checking temporal properties of runs As suggested above, the technique used in Theorem 3.2 allows one to verify certain temporal properties of runs. Consider the set $\mathcal{T}_{past\text{-}input}$ of temporal sentences of the form $\forall \vec{x} \varphi(\vec{x})$ where φ is a boolean combination of literals over **output**, **db** and **state**. A run satisfies this sentence if the sentence is verified at every stage of the run for the current output, database, and state relations. Note that a state atom of the form *past-R*(u) holds if $R(u)$ is has been input sometime in the past, which allows making temporal statements involving past inputs. For example, the statement "*deliver(x)* cannot be output unless *pay(x,y)* has been previously input, where *price(x,y)* is in the database" can be specified as the $\mathcal{T}_{past\text{-}input}$ sentence

$$\forall x \forall y[(deliver(x) \wedge price(x,y)) \rightarrow \ past\text{-}pay(x,y)].$$

Using a slight extension of the technique in Theorem 3.2 one can show:

Theorem 3.3 Given a Spocus relational transducer T and a sentence ψ in $\mathcal{T}_{past\text{-}input}$, it is decidable in NEXPTIME (NP if the number of variables in T is fixed) whether every run of T satisfies ψ.

We next consider problems involving the relationship between different transducers.

3.3 Containment of Spocus transducers

We consider here relationships between the runs of two Spocus transducers. Recall that a transducer T_1 contains a transducer T_2 (for a database D), if every log of T_2 with D is also a log of T_1 with D. The problem is undecidable in general, as shown next.

Theorem 3.4 Containment of Spocus transducers is undecidable (for D fixed or not).

Proof: (sketch) The proof is by reduction of the implication problem for functional and inclusion dependencies. The idea is reminiscent of the construction in the previous section, but is much more intricate due to the absence of projection in state rules of Spocus transducers. Suppose F and G are sets of IncDs and FDs over R. The construction involves generating in the state an arbitrary instance over R, and its projections relevant to the IncDs in F and G. This is achieved by inputting one tuple of R and its projections at each step of a run. The difficulty is to make sure the input in fact generates an instance R and its projections, using observations of the run provided by output relations. Violations of the IncDs and FDs can be verified using output rules that use R and its projections. To check that $F \models G$, we build transducers $T_{F,G}$ and T_G where $T_{F,G}$ checks violations of F and G, and T_G checks violations of G alone. Then $F \not\models G$ iff $T_{F,G} \supseteq T_G$. □

Fortunately, there is a special case of practical interest when the above problem becomes decidable. Suppose a Spocus transducer T_1 is given, and another transducer T_2 is constructed by augmenting T_1 with additional inputs and outputs. T_2 can be viewed as a customized version of T_1 (much like FRIENDLY is a customized version of SHORT). The proposed customization can be accepted as long as the logs of the runs of T_2 are still valid runs of T_1. This turns out to be decidable. More precisely, we can show:

Theorem 3.5 Given Spocus transducers T_1, T_2 with input schemas $\mathbf{in}_1$ and $\mathbf{in}_2$ where $\mathbf{in}_1 \subseteq \mathbf{in}_2$, and the same log schema which is full for T_1 (i.e. $\mathbf{in}_1 \subseteq \mathbf{log}$), it is decidable in NEXPTIME (NP if the number of variables in T_1 and T_2 is fixed) whether $T_1 \supseteq T_2$.

Similarly to previous results, the proof involves showing that if $T_1 \not\supseteq T_2$, then there is a "short and small" input sequence $\mathcal{I}$ such that $log_{T_1}(\mathcal{I}) \not\supseteq log_{T_2}(\mathcal{I})$.

As a consequence of Theorem 3.5, containment of Spocus transducers with full log is decidable:

Corollary 3.6 Given Spocus transducers T_1 and T_2 over the same schema and with full log, it is decidable in NEXPTIME (NP if the number of variables in T_1 and T_2 is fixed) whether $T_1 \supseteq T_2$ (for the database fixed or not).

As mentioned above, Theorem 3.5 is important to verify that customization is correct. An alternative to verification is to provide sufficient syntactic conditions for a customized program to preserve validity of the logs. A natural possibility is to allow adding inputs, outputs, and new rules, as long as the log is syntactically unaffected by the new inputs (i.e. there is no path from new inputs to relations in the log in the dependency graph of the program). For example, FRIENDLY can be obtained from SHORT in this manner.

So far, we considered no restrictions on input sequences. The temporal restrictions we studied, such as those expressed by $\mathcal{T}_{past-input}$, state that if something was output, then some pattern of inputs must have occurred in the past. This reflects the fact in our model outputs are driven by inputs, which are unrestricted. Indeed, inputs may arrive in any order. While this makes sense in some situations, in other applications one can clearly distinguish between valid and invalid sequences of inputs. For example, it may make sense to require that *order*(x) must be input before *pay*(x,y). We consider next a mechanism for specifying such restrictions, via the notion of *error-free run* .

3.4 Controlling input sequences

The basic transducer model can be enriched in various ways in order to accept only certain sequences of inputs, much like transducers in language theory can also be used as acceptors. We mention three ways to do this:

1. Define a distinguished output relation *error*. A run is valid if it is *error-free*, that is no output contains a literal over *error*.
2. Define a distinguished output relation *OK*. A run is valid if *every* output set in the sequence contains the literal *OK*.
3. Define a distinguished output relation *accept*. A run is valid iff it is finite and the last output set contains *accept*.

Perhaps surprisingly, the three mechanisms above are incomparable for Spocus transducers. For example, (1) allows enforcing natural restrictions such as *order*(x) must be input before *pay*(x,y). It turns out that such restrictions cannot be enforced by (2) or (3). On the other hand, (2) allows enforcing restrictions such as *every input set in a run must contain at least one new input*. This cannot be enforced by (1) or (3). A subtlety is that the comparison is affected by whether or not the log is full. For instance, if we allow unlogged inputs, the set of valid logged input sequences defined using (2) can also be defined by (1).

In the present paper, we focus on error-free runs since this allows specifying many restrictions of practical interest.

Enforcing properties of error-free runs As suggested above, using error-freeness to validate runs allows one to impose significant temporal properties on input sequences. To make this more precise, consider the set $\mathcal{T}_{sdi}$ of conjunctions of sentences of the form

$$\forall \vec{x}[\varphi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x}) \ \rightarrow \ \psi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x})]$$

where $\varphi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x})$ is a conjunction of literals over **state, db, in** with all variables $\vec{x}$ occurring in positive literals , and $\psi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x})$ is a disjunction of positive literals over **state, db, in** whose variables are among the $\vec{x}$. A run satisfies a sentence such as above iff the sentence is satisfied at every transition by the current state, database, and input. Examples of interesting filtering on runs that can be specified by sentences in $\mathcal{T}_{sdi}$ are as follows:

1. if x was ordered, x costs y and x was not previously paid, then the next valid input is to pay x or cancel the order:

$$\forall x \forall y[(past\text{-}order(x) \wedge price(x,y) \wedge \neg past\text{-}pay(x,y)) \rightarrow (pay(x,y) \vee cancel(x))]$$

2. if the amount y is paid for item x then x must have previously been ordered and y must be the correct price

$$\forall x \forall y[pay(x,y) \rightarrow (price(x,y) \wedge past\text{-}order(x))]$$

3. if the purchase of x is cancelled then x was previously ordered

$$\forall x[cancel(x) \rightarrow past\text{-}order(x)]$$

It turns out that such restrictions can be enforced in error-free runs. This indicates that Spocus transducers have considerable specification power, despite their simplicity. Indeed, one can show the following:

Theorem 3.7 For every formula $\psi \in \mathcal{T}_{sdi}$, there exists a Spocus transducer T such that the input sequences of its error-free runs are precisely those satisfying ψ.

Another useful way to understand the specification power of error-free runs is to consider the case when outputs are propositional. To further simplify things, consider such transducers where at most one proposition is output at each step, called *propositional-output transducers*. Output sequences can then be viewed as words over the finite alphabet of output propositions. Consider the language $Gen_{error\text{-}free}(T)$ consisting of all words output by T for some error-free finite run. We can show the following rather surprising result:

Theorem 3.8 A language L over alphabet Σ equals $Gen_{error\text{-}free}(T)$ for some propositional-output Spocus transducer iff L is a prefix-closed recursively enumerable language.

Proof: Clearly, each language $Gen_{error\text{-}free}(T)$ is prefix closed and recursively enumerable (r.e.) For the converse, suppose L is a prefix-closed r.e. language accepted by some Turing machine M. The idea is to build a Spocus transducer T whose *error* rules enforce that the sequence of inputs encodes consecutive configurations of a computation of M on an input word w. The encoding is quite intricate due to the inflationary nature of the state relations of T. If an accepting state is reached in the computation of M, T starts outputting w one letter at a time. Outputting the entire w requires an input sequence of appropriate length, short of which a prefix of w is output. □

An alternative to the prefix-closed requirement used above is to consider languages of the form $L\#$ where L is r.e. and $\#$ is an end-marker.

Verifying properties of error-free runs A natural question at this point is whether it can be *verified* whether the error-free runs of a Spocus transducer T satisfy a given sentence in $\mathcal{T}_{sdi}$. The problem is undecidable, but can be solved in the interesting case when *error* is defined by a set of rules where negation is not used on state literals. We state the undecidability result first.

Theorem 3.9 It is undecidable, given a Spocus transducer T and a sentence ψ in $\mathcal{T}_{sdi}$, whether every error-free run of T satisfies ψ.

Proof: The proof is similar to that of Theorem 3.4, by reduction of implication of functional and inclusion dependencies. □

Theorem 3.10 Given a sentence $\psi \in \mathcal{T}_{sdi}$ and a Spocus transducer T such that no negative state literal occurs in rules defining *error*, it is decidable in NEXPTIME (NP if the number of variables in T is fixed) whether every error-free run of T satisfies ψ.

Proof: The proof technique is similar to the previous decidability results: reduce the question to satisfiability of an $\exists^*\forall^*$FO sentence over a given schema. To fix the schema, it is enough to observe that only runs of length bounded by the number of occurrences in σ of **state** literals need be considered. □

Next, we compare transducers as acceptors, using their error-free runs. Containment of error-free runs turns out to be undecidable, even with full log.

Theorem 3.11 Given transducers T_1 and T_2 with the same schema, it is undecidable whether each error-free run of T_1 is also an error-free run of T_2, even with full log.

Proof: Similar to the proof of Theorem 3.9 □

However, decidability is obtained with various restrictions. Similarly to Theorem 3.10, we can show:

Theorem 3.12 Given transducers T_1 and T_2 with the same schema and full log such that no negative state literal occurs in rules defining *error* in T_1 or T_2, it is decidable in NEXPTIME (NP if the number of variables in T_1 and T_2 is fixed) whether every error-free run of T_1 is an error-free run of T_2.

4 Conclusion

Relational transducers were introduced to formally capture business models. The restricted Spocus transducers were put forward as a candidate model with several desirable features: ease of understanding and declarativeness of specification; decidability of various question concerning verification; and, ability to capture a wide range of business models of practical interest.

Many questions remain unanswered. For instance, it would be desirable to identify reasonable restrictions under which log validation is in PTIME. With respect to customization, one would like to be able to verify log containment under less restrictive conditions than the ones we impose. An alternative is to exhibit a set of rules for modifying relational transducers which preserve validity of logs. The goal is to provide the user with a tool-box facilitating sound customization of business models.

We argued that Spocus transducers capture a significant class of business models. We partly substantiated the claim by results on the ability of such transducers to specify valid sequences of inputs and outputs. It would be interesting to actually implement business models based on the Spocus framework to further validate the approach. Many problems need to be addressed to make the approach practical. For instance, an important issue is the optimization of the computation of state transitions, for which we can take advantage of incremental update techniques. Similarly, the management of triggers in active databases is clearly relevant, since relational transducers basically carry out a form of immediate triggering.

Perhaps the most challenging remaining issue is that of the interactions between relational transducers specifying business models of participants in a complex exchange. Such transducers can be combined in many ways, e.g. by having outputs of some transducers be input to other transducers, or having them share state relations. This raises new issues related to the verification of an interacting system of business models, including its overall consistency, detecting and resolving deadlock situations, etc. We plan to investigate such questions in future work.

Acknowledgments We would like to thank Al Aho and Alberto Mendelzon for discussions on this topic.

References

[AD97] S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views, 1997. available by http.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading-Massachusetts, 1995.

[BGG97] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer, Berlin, Heidelberg, 1997.

[BS28] P. Bernays and M. Schönfikel. Zum entscheidungsproblem der mathematischen logik. *Math. Annalen*, 99:342–372, 1928.

[CV85] A. K. Chandra and M. Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM J. on Computing*, 14(3):671–677, 1985.

[Eme91] E.A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 997–1072. Elsevier, 1991.

[FAY97] B. Fordham, S. Abiteboul, and Y. Yesha. Evolving databases: An application to electronic commerce. In *International Database Engineering and Applications Symposium (IDEAS)*, Montreal, 1997.

[Gur94] Y. Gurevich. *Evolving Algebras 1993: Lipari Guide, Specification and Validation Methods*. E. Börger, Oxford University Press, 1994.

[Lew80] H. Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21:317–353, 1980.

[Mil80] R. Milner. A calculus of communicating systems. In *LNCS*, volume 92. Springer-Verlag, 1980.

[Mil91] R. Milner. Operational and algebraic semantics of concurrent processes. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1201–1242. Elsevier, 1991.

[Mit83] J. C. Mitchell. Inference rules for functional and inclusion dependencies. In *Proc. ACM Symp. on Principles of Database Systems*, pages 58–69, 1983.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Theoretic Computer Science*, volume 104, pages 167–183. Springer-Verlag, Berlin, 1981.

[PV97] P. Picouet and V.Vianu. Semantics and expressiveness issues in active databases. *J. of Computer and System Sciences*, 1997. to appear.

[Ram30] F. Ramsey. On a problem in formal logic. *Proc. of the London Math. Society*, 2nd Series(30):264–286, 1930.

[Rei83] W. Reisig. Petri nets. In *EATCS Monongraph on Theoretical Computer Science*, volume 4. Springer, Berlin, 1983.

[SR86] M. Stonebraker and L. Rowe. The design of Postgres. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 340–355, 1986.

[Ull89] J. D. Ullman. Bottom-up beats top-down for datalog. In *Proc. ACM Symp. on Principles of Database Systems*, pages 140–149, 1989.

[WC95] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, San Francisco, California, 1995.

[work93] Special issue on workflow and extended transaction systems. *Data Engineering Bulletin*, 16(2), 1993.

[YA96] Yelena Yesha and Nabil Adam. Electronic commerce: An overview. In Nabil Adam and Yelena Yesha, editors, *Electronic Commerce*. Lecture Notes in Computer Science, Springer-Verlag, 1996.