Relational Transducers for Electronic Commerce

Serge Abiteboul
INRIA-Rocquencourt
Serge.Abiteboul@inria.fr

Victor Vianu[1]
University of California, San Diego
vianu@cs.ucsd.edu

Brad Fordham
Oracle Corporation
bfordham@us.oracle.com

Yelena Yesha
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.

- *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*) transducer. 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 Σ_2^p if the schema 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, PV98], although the questions we consider are quite different. The logic background is predicate logic for the static aspect and temporal logic [Eme91] for the temporal aspect. Since in some sense we are dealing with interacting processes, our work could be viewed in the more general context of 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. An "attribute-centric" approach to declarative specification of workflows is developed in [H+99]. The emphasis is on workflows where attribute values for a given object are collected or computed in the course of the workflow. The workflow is specified using a rule-based language called Vortex.

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], now called abstract state machines, 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 hard to understand by users. This is the primary 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 on verification of single transducers and of containment of transducers. In Section 4 we consider a mechanism to control *inputs* to Spocus transducers using the notion of error-free run. We consider the expressiveness of such transducers and revisit some of the verification questions concerning single transducers and comparison of transducers. 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 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: 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. The rules in the example have the obvious semantics. 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. 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.)

The last component of the schema consists of the *log* relations, which are a subset of the input and output relations. 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 a run to the log relations is simply called the *log of the run*.

This 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.

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

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 finding a minimum log that is sufficient to verify correctness of transactions. It is easy to see that in the previous transducer, one can remove the relation *deliver* from the log without losing any information. 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;
  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, past-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 **schema** be a transducer schema. A *relational transducer* over **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, **state** (**out**). The mapping σ is called the *state function* and ω the *output function*.

Given a sequence $I_1, ..., I_n$ over **in** (called an *input sequence*), and a database instance D of **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 **in** relations is to describe the inputs from users of the system. The **db** relations represent a database used by the system (possibly very large and external). The **state** relations represent the information that the system remembers from its current run. The **out** relations capture the reactions of the system, and the **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 σ 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 :- $A_1, \ldots, A_n$ 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 provide 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.

Example The following propositional 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 a Spocus transducer cannot control its input, but only its output. For instance, we cannot prevent A to be input several times, but, using *past-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 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 case 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).

The Bernays-Schönfinkel prefix class By way of preliminaries, we note that most of the decidability results in the paper are shown by reduction to finite satisfiability of FO sentences with relational vocabulary, constants, and equality, of the form

$$\exists x_1 \ldots \exists x_k \forall y_1 \ldots \forall y_m \ \varphi$$

$k \geq 0, m \geq 0$, 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]. The decidability follows from a straightforward observation: if a sentence $\exists x_1 \ldots \exists x_k \forall y_1 \ldots \forall y_m \ \varphi$ has a model, then it has a model with $max(1, k)$ elements. The complexity of the decision procedure was investigated in [Lew80], and it was proven that the problem is complete in NEXPTIME. If the arity of relations in the signature is bounded, then the problem is complete in Σ_2^p (the class NP^{NP} in the polynomial hierarchy), 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 database instance D and a log $\mathcal{L}$, it is decidable in NEXPTIME (Σ_2^p if the schema of 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. We next describe this sentence.

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 $\mathcal{I}$. Consider first input relations. Suppose L_j specifies that input relation R_j consists of a certain set of tuples. A sentence stating this says that every tuple in the log belongs to R_j, and that every tuple in R_j belongs to the log. Saying that every tuple in the log belongs to R_j is done by a conjunction of sentences, one for each tuple in the log. Each such sentence is in $\exists^*$FO. For example, if the log specifies that a tuple $\langle a_1, \ldots, a_k\rangle$ belongs to R_j (of arity k), this is stated by the $\exists^*$FO sentence

$$\exists x_1 \ldots \exists x_k \ (R_j(x_1, \ldots, x_k) \wedge \bigwedge_{i=1}^{k} (x_i = a_i)).$$

Saying that the input relation is included in the log requires a $\forall^*$FO sentence. For example, if the log specifies that R_j contains precisely $\{\langle a_1, \ldots, a_k\rangle, \langle b_1, \ldots, b_k\rangle\}$, the inclusion of R_j in the log is specified by:

$$\forall x_1 \ldots \forall x_k \ [R_j(x_1, \ldots, x_k) \ \rightarrow \ ((x_1 = a_1 \ \wedge \ldots \wedge \ x_k = a_k) \vee (x_1 = b_1 \ \wedge \ldots \wedge \ x_k = b_k))].$$

Now consider the more interesting case of the output relations of the log. Again, stating that the tuples in the log belong to an output relation can be done by a $\exists^*$FO sentence. For example, let us say that R of arity k is such an output relation. Suppose R is defined by a single output rule (the general case is an easy extension), of the form:

$$R(x_1, \ldots, x_k) \ := \ A_1, \ldots, A_m$$

where (by definition of output rules) each A_v is of the form $(\neg)Q(\vec{z})$ or $x \neq y$ where Q is an input, state, or database relation, and each variable occurs in some positive literal. Then the content of R_j is defined by a formula on the extended relational schema as follows. Let $\vec{y}$ be the variables occurring in $A_1, \ldots, A_m$ other than $x_1, \ldots, x_k$. The formula is

$$\varphi(x_1, \ldots, x_k) = \exists \vec{y}(\bar{A}_1 \wedge \ldots \wedge \bar{A}_m)$$

where $\bar{A}_v$ is defined as follows:

- if A_v involves a database relation or is $x \neq y$, then $\bar{A}_v$ is simply A_v.
- if A_v is of the form $(\neg)Q(\vec{z})$ where Q is an input relation, then $\bar{A}_v$ is $(\neg)Q_j(\vec{z})$.
- if A_v is of the form $(\neg)\mathit{past}\text{-}Q(\vec{z})$, then $\bar{A}_v$ is $(\neg)(Q_1(\vec{z}) \vee \ldots \vee Q_{j-1}(\vec{z}))$.

Note that $\varphi(x_1, \ldots, x_k)$ is an $\exists^*$FO formula. The remainder is similar to the case of input relations. Saying that every tuple in the log belongs to R_j is done by a conjunction of sentences, one for each tuple in the log. Each such sentence is of the same form as for input relations, except that $R_j(x_1, \ldots, x_k)$ is replaced by $\varphi(x_1, \ldots, x_k)$ (the result is an $\exists^*$FO sentence). Saying that the output relation is included in the log requires a $\forall^*$FO sentence. For example, if the log for R_j is $\{\langle a_1, \ldots, a_k\rangle, \langle b_1, \ldots, b_k\rangle\}$, the inclusion of R_j in the log is stated by:

$$\forall x_1 \ldots \forall x_k \; [\varphi(x_1, \ldots, x_k) \;\rightarrow\; ((x_1 = a_1 \;\wedge \ldots \wedge\; x_k = a_k) \vee (x_1 = b_1 \;\wedge \ldots \wedge\; x_k = b_k))].$$

Since $\varphi(x_1, \ldots, x_k)$ is an $\exists^*$FO sentence, the above is a $\forall^*$FO sentence.

Altogether, the sentence specifying the log is a conjunction of $\exists^*$FO and $\forall^*$FO sentences. This can be written in prenex form as an $\exists^*\forall^*$FO sentence, whence the decidability of log validity in NEXPTIME (Σ_2^p if the schema of T is fixed; note that the replication of the input schema does not affect the complexity, since the arity of relations remains unchanged). □

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 next consider the impact of restricting or extending Spocus transducers on checking log validity:

(1) Spocus Restrictions: Log validation remains expensive even for restricted Spocus transducers. Thus, log validation is NP-hard even if output relations are defined by conjunctive queries over the state relations alone. This is because view consistency is NP-hard for views defined by conjunctive queries [AD98]. Other problems however become simpler 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) remain 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. We show this by reduction of the implication problem for functional and inclusion dependencies (FDs and IncDs) which is undecidable [CV85, Mit83]. The idea of the reduction is the following. Given sets F, G of FDs and IncDs over some relation R, we construct a transducer with extended state rules that does the following: (i) on input R, the transducer stores 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 $\langle \emptyset, \{\mathit{violation}\text{-}G\}\rangle$ 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).
```

More formally, we have:

Proposition 3.1 *Log validity is undecidable for Spocus transducers extended by allowing projections in state rules.*

Proof: This can be shown by reduction of the implication problem for functional and inclusion dependencies. We recall the problem next.

Let R be a relation of arity n. A *functional dependency* (FD) over R is an expression of the form $X \rightarrow j$ where X is a subset of $[1..n]$ and j is in $[1..n]$. The functional dependency $X \rightarrow j$ is usually denoted $i_1...i_m \rightarrow j$ assuming some ordering $i_1...i_m$ of the elements of X (e.g., $13 \rightarrow 2$). An instance I of relation R satisfies $X \rightarrow j$ if for each tuple u, v in I, if they agree on X, i.e., $\pi_X(u) = \pi_X(v)$, they also agree on j, $u(j) = v(j)$. An *inclusion dependency* (IncD) over R is an expression of the form $i_1...i_m \subseteq j_1...j_m$ where for each k, i_k and j_k are in $[1..n]$. An instance I of relation R satisfies $i_1...i_m \subseteq j_1...j_m$ if for each tuple u in I, there exists a tuple v in I such that for each k, $u(i_k) = v(j_k)$.

A set of FDs and IncDs F over R implies another set G of such dependencies ($F \models G$) if each instance of R satisfying F also satisfies G. The implication problem for FDs and IncDs is known to be undecidable [CV85, Mit83]. (See [AHV95] for more on dependencies.)

We reduce the implication problem for FDs and IncDs to the log validity problems for Spocus transducers extended by allowing projections in state rules. Let R be a relation of arity n and F, G two sets of FDs and IncDs over R. We construct a transducer T with input R as follows:

state relations The state relations are *past-R* and *past-*$R_{j_1...j_m}$ for each $j_1, ..., j_m$ such that

$$i_1...i_m \subseteq j_1...j_m \text{ in } F \text{ or } G \text{ for some } i_1...i_m.$$

state rules The state rules fill in *past-R* and its projections. [This is the single place we use non Spocus rules.] More precisely, T has the rules:

$$\begin{array}{lll} \textit{past-}R(x_1, ...x_n) & \text{+:- } R(x_1, ..., x_n) & \\ \textit{past-}R_{j_1...j_m}(x_{i_1}, ..., x_{i_m}) & \text{+:- } R(x_1, ..., x_n) & \text{for each } \textit{past-}R_{j_1...j_m} \text{ relation.} \end{array}$$

output relations The output relations are $violF$ and $violG$ of arity zero. The log consists of the output relations only.

output rules The output rules detect violations of F and G. For each FD $i_1...i_m \rightarrow j$ in F, we have the rule:

$$violF \text{ :- } \textit{past-}R(x_1, ...x_n), \textit{past-}R(y_1, ..., y_n), x_{i_1} = y_{i_1}, ..., x_{i_m} = y_{i_m}, x_j \neq y_j.$$

For each IncD $i_1...i_m \subseteq j_1...j_m$, we have the rule:

$$violF \text{ :- } past\text{-}R(x_1, ..., x_n), \neg past\text{-}R_{j_1...j_m}(x_{i_1}, ..., x_{i_m}),$$

and similarly for G and $violG$.

Consider the log sequence $\mathcal{L} = \langle \emptyset, \{violG\} \rangle$. We show that $\mathcal{L}$ is valid for T if and only if $F \not\models G$.

Suppose first $F \not\models G$. Let I be an instance satisfying F and not G. (Such an instance exists since $F \not\models G$.) On input $\langle I, \emptyset \rangle$, T would produce $\mathcal{L}$, so $\mathcal{L}$ is valid. (The first instance of the log for T is always empty because all state relations are empty to start. And the second will detect the violation of G but not F in I.)

Conversely, suppose $\mathcal{L}$ is valid for T. Let $\langle I, I' \rangle$ be the input that produced $\mathcal{L}$. Let $\langle S, S' \rangle$ be the state sequence. Clearly, S consists of I and some projections of I. Since $violG$ is derived and not $violF$, one can conclude that I satisfies F and not G, so $F \not\models G$. $\Box$

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. 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 (Σ_2^p if the schema of 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, note that only runs of length two need be considered. Indeed, consider a Spocus transducer T and an input sequence $\mathcal{I} = I_1 \ldots I_n$. Since outputs depend only on the current input, the database, and the state relations (containing the union of all previous inputs), the last output in the run of T on $\mathcal{I}$ is the same as the last output on the run of T on the sequence of two inputs $(\cup_{1 \leq i < n} I_i), I_n$. Next, the problem is reduced to that of satisfiability of an $\exists^*\forall^*$FO sentence over a schema consisting of two copies of the input. $\Box$

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, 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 (Σ_2^p if the schema of T is fixed) whether every run of T satisfies ψ.

Proof: Let T be a Spocus relational transducer and ψ a sentence in $\mathcal{T}_{past\text{-}input}$. By definition, every run of T satisfies ψ iff there is no run violating ψ at some point in the run. This is equivalent to unsatisfiability of $\neg\psi$. Now $\neg\psi$ is a sentence of the form $\exists\vec{x}\varphi(\vec{x})$ where $\varphi(\vec{x})$ is a Boolean combination of literals over **output**, **db** and **state**. Since existential quantification distributes over disjunction, testing unsatisfiability of $\exists\vec{x}\varphi(\vec{x})$ can be reduced to testing unsatisfiability of a set of sentences of the form $\exists\vec{x}\psi(\vec{x})$ where ψ is a conjunction of literals over **output**, **db** and **state**. By modifying T so that **db** and **state** are made part of the output, this reduces to testing unsatisfiability of a sentence of the form $\exists\vec{x}(A_1 \wedge \ldots \wedge A_n)$ where the A_i are literals over **output**. By Theorem 3.2 this can be done in NEXPTIME (Σ_2^p if the schema of T is fixed). The entire procedure can be performed within the same complexity. $\Box$

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 (even with fixed database).

Proof: The proof is by reduction of the implication problem for functional and inclusion dependencies. The idea is reminiscent of the construction in the proof of Proposition 3.1. It is more intricate due to the absence of projections in state rules of Spocus transducers. The difficulty is to make sure the input causes the state relations to contain an instance and its appropriate projections. This is done using observations of the run provided by output relations.

Suppose again that R is a relation of arity n and that F, G are two sets of FDs and IncDs over R. We first build a transducer $T_{F,G}$ that constructs instances of R and its projections needed to check violations of F or G. These are constructed by inputs that insert one tuple at a time in R and its projections. We call such input sequences well-formed. Violations of F and G are signaled by outputs $violF$ and $violG$. If $F \models G$, $violG$ is never be output without $violF$ in a run on a well-formed input sequence. The transducer also has rules that check if the input is well-formed. This is done using two output predicates *ok* and *error*, so that a run is well-formed iff *ok* is output at every step and *error* is never output.

Next, we build a much simpler transducer T that mimics the behavior of $T_{F,G}$ in the case when $F \models G$. First, the transducer can output $\{ok\}, \{ok, violF\}, \{ok, violF, violG\}$. This mimics the outputs of $T_{F,G}$ on well-formed inputs when $F \models G$. Additionally, if ok is not output at some stage or if $error$ is output at least once, T can also output $violG$ without $violF$. This corresponds to runs of $T_{F,G}$ on inputs that are not well-formed, on which false violations of $F \models G$ may be detected. By construction, $F \models G$ iff $T_{F,G} \subseteq T$.

We next present the construction of $T_{F,G}$ and T in more detail. We begin with $T_{F,G}$. As discussed above, we are mostly interested in the output of $T_{F,G}$ when tuples are input into R one at a time, although we clearly cannot impose such restrictions on inputs. Transducer $T_{F,G}$ is defined as follows:

input $\{R\} \cup \{R_{j_1 \ldots j_m}\} \cup \{A_i\}$

We have, as input relations, R and $R_{j_1 \ldots j_m}$ for each $j_1, \ldots, j_m$ such that

$$i_1 \ldots i_m \subseteq j_1 \ldots j_m \text{ in } F \text{ or } G \text{ for some } i_1 \ldots i_m.$$

For each i in $[1..n]$, A_i is also an input relation of arity 1. (For each i, A_i will contain the ith coordinate of the input tuple.)

state $\{R\} \cup \{R_{j_1 \ldots j_m}\}$

Recall that the state rules simply cumulate all past inputs for these relations.

output, log $\{violF, violG, ok, error\}$

All these relations are both output and log and are 0-ary.

Relations $violF$ and $violG$ are defined using exactly the same rules as in Proposition 3.1. (A subtlety is that the $past\text{-}R_{j_1 \ldots j_m}$ relations are now accumulating input tuples instead of accumulating projections of input tuples.)

Relation $error$ and ok controls whether the input is "well-formed". An input is well-formed if exactly one tuple at a time is inserted into R, together with its projections on $R_{j_1 \ldots j_m}$ and A_i. The following output rules are used for $error$:

1.	$error$:- $A_i(x), A_i(y), x \neq y$	for each i
2.	$error$:- $R(x_1, \ldots, x_n), \neg A_i(x_i)$	for each i
3.	$error$:- $A_1(x_1), \ldots, A_n(x_n), \neg R(x_1, \ldots, x_n)$	
4.	$error$:- $R(x_1, \ldots, x_n), \neg R_{j_1 \ldots j_m}(x_{i_1}, \ldots, x_{i_m})$	for each $R_{j_1 \ldots j_m}$
5.	$error$:- $R_{j_1 \ldots j_m}(x_{j_1}, \ldots, x_{j_m}), R_{j_1 \ldots j_m}(y_{j_1}, \ldots, y_{j_m}), x_{j_k} \neq y_{j_k}$	for each $R_{j_1 \ldots j_m}, k$

The single rule for ok controls that no A_i relation is empty at some step. It is given by:

$$ok \text{ :- } A_1(x_1), \ldots, A_n(x_n)$$

We first show that:

(*) An input is well-formed if and only if (i) $error$ is never generated and (ii) ok is generated at each step.
(+) On well-formed inputs, the state relations contain at every step, an instance of R and its proper projections.
(++) On well-formed inputs, if $F \models G$, then at each step, $T_{F,G}$ outputs either $\{ok\}$, $\{ok, Fviol\}$ or $\{ok, Fviol, Gviol\}$.

Suppose that an input sequence is well-formed. It is immediate to see that *error* is never generated and that *ok* is derived at each step. Conversely, suppose that the input sequence satisfies (i) and (ii). By (1) and (ii), each A_i has exactly one value. By (2) and (3), R contains exactly one tuple, the cross product of the A_i. By (4) and (5), each $R_{j_1 \ldots j_m}$ contains the proper projection of R. Thus the input sequence is well-formed. Hence, by construction, the state relations contain at every step, an instance of R and its proper projections. Thus (+) holds and (++) follows.

We now construct a transducer T that simulates the output of $T_{F,G}$ assuming that $F \models G$. All inputs, outputs, and state relations are 0-ary (propositional). No database relations are used. The schema of T is as follows:

input { *sim_F, sim_G, sim_G', sim_error, sim_notok* }

state { *past-sim_error, past-sim_notok* }

output, log { *violF, violG, ok, error* }

These relations are both in the output and in the log.

The following output rules are used by T to simulate $T_{F,G}$ on well-formed input sequences (assuming $F \models G$):

$$\begin{aligned} violF &\text{ :- } sim\_G \\ violG &\text{ :- } sim\_G \\ violF &\text{ :- } sim\_F. \end{aligned}$$

Additional rules simulate $T_{F,G}$ on non well-formed inputs where *error* is produced at some step:

$$\begin{aligned} error &\quad \text{:- } sim\_error \\ violG &\quad \text{:- } past\text{-}sim\_error, sim\_G' \end{aligned}$$

Note that we can derive arbitrarily *error* facts. Observe also that when *error* has been derived once, we have a rule that allows to produce *violG* without producing *violF*.

Finally, we have rules to simulate $T_{F,G}$ on non well-formed inputs where *ok* is absent at some step:

$$\begin{aligned} ok &\quad \text{:- } \neg sim\_notok \\ violG &\quad \text{:- } past\text{-}sim\_notok, sim\_G' \end{aligned}$$

Note that we can produce arbitrarily *ok* outputs. To block the derivation of *ok*, we use the *presence* of *sim_notok*. Also observe that if *ok* was absent at one step (so *sim_notok* was in the input), the last rule allows to produce *violG* without producing *violF*.

We claim that $F \models G$ if and only if $T_{F,G} \subseteq T$. For suppose that $F \models G$. For well-formed input sequences, T can produce the same log as $T_{F,G}$, driven by the input relations *sim_F* and *sim_G*. Consider a non well-formed input sequence. On well-formed prefixes, T simulates $T_{F,G}$ as for well-formed input sequences. At the first step where the input sequence violates well-formedness, either *error* is output of *ok* is missing from the output. We use *sim_error* or *sim_notok*, respectiveley, in the input sequence of T at that step. It is then easy to simulate $T_{F,G}$ on the remainder of the run using *sim_F* and *sim_G'*.

Conversely, suppose that $F \not\models G$. Let I be an instance that satisfies F and not G. Consider $T_{F,G}$ on a well-formed input sequence constructing I one tuple at a time. Then the final output will contain *violG* and not *violF*. Clearly, this log is not a valid log for T. Thus $T_{F,G} \not\subseteq T$. $\Box$

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 (Σ_2^p for fixed schema) whether $T_1 \supseteq T_2$.

Proof: Because $\mathbf{in}_1 \subseteq \mathbf{in}_2$, $T_1 \not\supseteq T_2$ iff there exists some input sequence $\mathcal{I}$ over $\mathbf{in}_2$ such that the log of T_2 on $\mathcal{I}$ differs from the log of T_1 on the same input restricted to $\mathbf{in}_1$. Furthermore, as in the proof of Theorem 3.2, it is easily seen that if such input sequence $\mathcal{I}$ exists, then there also exists such a sequence of length two. Testing that the log of T_2 differs from that of T_1 on an input sequence of length two is expressed by a sentence σ in $\exists^*\forall^*$FO, over a schema obtained by taking two copies of $\mathbf{in}_2$. Thus, testing if $T_1 \not\supseteq T_2$ is reduces to testing satisfiability of a $\exists^*\forall^*$FO sentence over this schema. This has complexity NEXPTIME (Σ_2^p for fixed schema). $\Box$

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 (Σ_2^p for fixed schema) whether $T_1 \supseteq T_2$ (with the database fixed or not).

As mentioned above, Theorem 3.5 is important in order 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 that 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* .

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). The proof of Theorem 3.4 provides an instructive illustration of the power of (1) in conjunction with (2).

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

4.1 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, we define the following rich set of sentences.

Definition: Let $\mathcal{T}_{sdi}$ consist 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

1. $\varphi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x})$ is a conjunction of literals over **state, db, in** with all variables $\vec{x}$ occurring in positive literals, and

2. $\psi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x})$ is a quantifier-free positive formula over **state, db, in** whose variables are among the $\vec{x}$.

A run *satisfies* a sentence in $\mathcal{T}_{sdi}$ if and only if the sentence is satisfied at every transition by the current state, database, and input.

Some examples of interesting properties of 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:

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

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 confirms that Spocus transducers have considerable specification power, despite their simplicity. Indeed, one can show the following:

Theorem 4.1 For every formula $\theta \in \mathcal{T}_{sdi}$, there exists a Spocus transducer T such that the input sequences of its error-free runs are precisely those satisfying θ.

Proof: Let θ be in $\mathcal{T}_{sdi}$. First observe that we may assume without loss of generality that θ consists of a single formula of the form:

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

with φ, ψ as in the definiion of $\mathcal{T}_{sdi}$. For suppose that θ has more than one conjunct. Then for each of the conjuncts C_i, there exists a transducer that detects violations of C_i. It is immediate to construct a transducer detecting violations of $\wedge C_i$. Furthermore, we may also assume that ψ is a disjunction of literals. For suppose that this is not the case. Then we can write ψ in conjunctive normal form:

$$\forall \vec{x}[\varphi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x}) \ \rightarrow \ (\psi_1(\vec{x}) \wedge ... \wedge \psi_m(\vec{x}))] \equiv$$
$$(\forall \vec{x}[\varphi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x}) \ \rightarrow \ \psi_1(\vec{x})]) \wedge ... \wedge (\forall \vec{x}[\varphi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x}) \ \rightarrow \ \psi_n(\vec{x})])$$

Each of the conjuncts can be verified separately.

So, let:

$$\theta = \forall \vec{x}[\varphi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x}) \ \rightarrow \ (L_1(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x}) \vee ... \vee L_m(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x})]$$

where each L_i is a positive literal. This sentence is violated when its negation holds, i.e., if at some step we have:

$$\exists \vec{x}[\varphi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x}) \wedge \neg L_1(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x}) \wedge ... \wedge \neg L_m(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x})]$$

Since the L_i are literals and φ is a conjunction of literals over **state, db, in** with all variables of $\vec{x}$ occurring in positive literals, this can be detected by the Spocus rule:

$$\mathit{error}\text{:- } \varphi(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x}), \neg L_1(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x}), \ldots, \neg L_m(\mathbf{state}, \mathbf{db}, \mathbf{in})(\vec{x}).$$

□

Another useful way to understand the specification power of error-free runs is to consider transducers with propositional output, where at most one proposition is output at each step of error-free runs. Let us call such transducers *propositional-output transducers*. Output sequences of finite error-free runs 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 a propositional output transducer T for some error-free finite run. We can show the following rather surprising result, which yields considerable insight into the power of Spocus transducers and provides a key technical tool:

Theorem 4.2 A language L over alphabet Σ equals $Gen_{error\text{-}free}(T)$ for some propositional-output Spocus transducer T 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 an r.e. language. We build a propositional-output Spocus transducer T such that $Gen_{error\text{-}free}(T)$ is the prefix closure of L. There exists a nondeterministic Turing machine M whose halting configurations on input ϵ contain on the tape exactly the words in L. We construct a Spocus transducer T whose *error* rules enforce that a sequence of inputs encodes consecutive configurations of a computation of M on input ϵ. The encoding is quite intricate due to the inflationary nature of the state relations of T. If a halting state is reached in the computation of M, T starts outputting the word w generated on the tape, 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.

We next outline the main steps in the construction of T. We may assume that M is a nondeterministic right-infinite tape Turing Machine generating L on input ϵ. We can also assume that in a halting configuration, the output word starts at the leftmost tape cell and the head of M is positioned at the same cell. A computation of M is simulated by inputting consecutive configurations of M in a designated input relation *tape* of T. The consecutive configurations are cumulated in the state relation *past-tape* corresponding to the input relation *tape*. Error rules are used to check each new configuration input in *tape* is a configuration obtained by a legal move of M from the most recent configuration stored in *past-tape*. Since state relations are inflationary (nothing is ever deleted), the different configurations must be timestamped so that the most recent configuration can be identified. The input relation *tape* of T encoding a timestamped configuration of M is of the form:

tape	*stamp*	$index_1$	$index_2$	*cell*	*state*
	α	0	1	c_1	s_1
	α	1	a_2	c_2	s_2
	α	a_2	a_3	c_3	s_3
			...		
	α	a_{n-1}	a_n	c_n	s_n

where α is the timestamp of the configuration, $0, 1, a_2, \ldots, a_n$ are distinct values indicating the order of the tape cells, $c_1, c_2, \ldots, c_n$ provide the content of the first n tape cells, and $s_1, s_2, \ldots, s_n$ indicate the position of the head and the current state as follows: if the head points at cell i and M is in state q then $s_j = 0$ for $j \neq i$ and $s_i = q$. For example, a configuration 011001 where the head points to the third cell and the state is q might be represented as follows:

tape	*stamp*	$index_1$	$index_2$	*cell*	*state*
	α	0	1	0	0
	α	1	17	1	0
	α	17	8	1	q
	α	8	5	0	0
	α	5	20	0	0
	α	20	6	1	0

The simulation of M by T has three main stages:

1. construct an encoding of the initial configuration of M, including a blank tape of arbitrary finite length;

2. simulate the computation of M until a halting state is reached;

3. output the word on the tape one letter at a time.

To move from one stage to the next in the desired order, T remembers the current stage using a unary input relation *stage* and its corresponding state relation *past-stage*. Starting stage (i) is signaled by inputting $stage(i)$. Checking that the right sequence of transitions is observed is done by error rules of the form

$$\begin{array}{lll} error & :- & stage(x), stage(x'), x \neq x' \\ error & :- & \neg stage(1), \neg stage(2), \neg stage(3) \\ error & :- & stage(i), past\text{-}stage(i+1) \qquad i \in \{1,2\} \\ error & :- & stage(i), \neg past\text{-}stage(i-1) \qquad i \in \{2,3\} \end{array}$$

Each of the stages uses its own input and state relations to achieve its goal. To avoid cross-talk among the different stages, it is helpful to impose that inputs irrelevant to the current stage be empty. This is easily done using error rules of the form

$$error \quad :- \quad stage(i), R(\vec{x}) \quad \text{for } i = 1,2,3$$

where R is an input relation irrelevant to stage i. Similarly, rules of T that are irrelevant to a given stage can also be inhibited. In the subsequent development we omit to include explicitly such control rules or clauses.

We now outline each of the stages in the simulation of M by T.

Stage 1 T begins by constructing in state relation *past-tape* an encoding of the initial configuration of M, including a blank tape of finite length, chosen arbitrarily. The tape of the machine is never extended throughout the simulation. If the length of the initial tape is insufficient, then the simulation fails and nothing is output. Moreover, the ordered indexes used to represent the initial blank tape are also used as timestamps of the configurations in the computation of M. Again, if the number of indexes is insufficient then the simulation fails and nothing is output.

The construction of the state relation *past-tape* encoding the initial configuration is achieved by inputting one tuple at a time into the input relation *tape*. The first pair of indexes consists of $\langle 0,1 \rangle$, the timestamp of the initial configuration is 0, the initial state is q_0, and the head is placed at the first cell. This first step is achieved by inputting $tape(0,0,1,b,q_0)$ (the blank symbol is denoted by b). To make sure this is indeed the first step in the computation, the following error rule is used:

$$error :- \neg past\text{-}stage(1), stage(1), \neg tape(0,0,1,b,q_0)$$

Subsequently, we wish to insert tuples of the form $tape(0,\alpha,\beta,b,0)$ where α is the last inserted index and β is a new index. To do this, we must keep track of the last inserted index. This is done using two additional unary input relations *index* and *oldindex*. When $tape(0,\alpha,\beta,b,0)$ is inserted, $index(\beta)$ and $oldindex(\alpha)$ should be inserted as well. If this is done, the difference between *past-index* and *past-oldindex* contains the current maximum.

The relations *index* and *oldindex* are initialized by inserting $index(0), index(1), oldindex(0)$ in the same first step when $tape(0,0,1,b,q_0)$ is inserted. This is checked by the rules:

$$\begin{array}{lll} error & \text{:-} & \neg past\text{-}stage(1), stage(1), \neg index(0) \\ error & \text{:-} & \neg past\text{-}stage(1), stage(1), \neg index(1) \\ error & \text{:-} & \neg past\text{-}stage(1), stage(1), \neg oldindex(0) \end{array}$$

We can use the following error rules to (partially) enforce that the inductive step in the construction works properly. It is easy to enforce that after the first step of stage (1), at most one tuple is inserted in any input relation, and any tuple inserted in *tape* is of the form $\langle 0, \alpha, \beta, b, 0\rangle$ (the corresponding rules are omitted). Rules (1-3) say that, if $tape(0,\alpha,\beta,b,0)$ is inserted, then α must be the previous maximum index and β must be new:

$$\begin{array}{llll} (1) & error & \text{:-} & tape(0,\alpha,\beta,b,0), \neg past\text{-}index(\alpha) \\ (2) & error & \text{:-} & tape(0,\alpha,\beta,b,0), past\text{-}oldindex(\alpha) \\ (3) & error & \text{:-} & tape(0,\alpha,\beta,b,0), past\text{-}index(\beta) \end{array}$$

Rules (4-6) say that $tape(0,\alpha,\beta,b,0)$ is inserted iff $oldindex(\alpha)$ and $index(\beta)$ are also inserted.

$$\begin{array}{llll} (4) & error & \text{:-} & tape(0,\alpha,\beta,b,0), \neg oldindex(\alpha) \\ (5) & error & \text{:-} & tape(0,\alpha,\beta,b,0), \neg index(\beta) \\ (6) & error & \text{:-} & oldindex(\alpha), index(\beta), \neg tape(0,\alpha,\beta,b,0) \end{array}$$

Rules (7-8) say that if $index(\beta)$ is input and α is the previous maximum, then $tape(0,\alpha,\beta,b,0)$ and $oldindex(\alpha)$ must also be input.

$$\begin{array}{llll} (7) & error & \text{:-} & index(\beta), past\text{-}index(\alpha), \neg past\text{-}oldindex(\alpha), \neg tape(0,\alpha,\beta,b,0) \\ (8) & error & \text{:-} & index(\beta), past\text{-}index(\alpha), \neg past\text{-}oldindex(\alpha), \neg oldindex(\alpha) \end{array}$$

Finally, rules (9-10) ensure that if $oldindex(\alpha)$ is inserted then α was the previous maximum.

$$\begin{array}{llll} (9) & error & \text{:-} & oldindex(\alpha), \neg past\text{-}index(\alpha) \\ (10) & error & \text{:-} & oldindex(\alpha), past\text{-}oldindex(\alpha) \end{array}$$

There remains one input combination that is undesirable but cannot be detected by error rules, when the input consists of $oldindex(\alpha)$ alone (and *tape*, *index* are empty). However, note that if such a combination is input then *past-index* and *past-oldindex* become equal and by rules (1-2) no tuple can be subsequently input into *tape* without generating an error. Thus, such an occurrence ends the construction of the tape. The simulation then proceeds with the tape constructed so far.

Stage 2 In stage 2, each input must provide in relation *tape* a complete configuration that can be obtained from the most recent one by a legal move of M. Configurations are timestamped using the ordered indexes found in *past-tape*. The newly input configuration must be the same as the most recent one except for the cell to which the head points, the preceding cell, or the following cell, depending on the move of M that is simulated. Correctness of the new configuration is checked by the error rules below. The first rule ensures that a unique timestamp is used in the input *tape*:

$$\begin{array}{llll} (1) & error & \text{:-} & tape(\beta,x,y,z,v), tape(\beta',x',y',z',v'), \beta \neq \beta' \end{array}$$

Rules (2-3) enforce that the ordered set of indexes used in the input is precisely the same as in previous configurations stored in *past-tape*. In these and other rules, Δ is the set of all possible values of attributes *cell* and *state* of *tape* (tape alphabet symbols including the blank symbol, and states of M):

$$\begin{array}{lll}(2) & error & \text{:-} \quad tape(\beta, x, y, z, v), past\text{-}tape(\alpha, x', y', z', v'), \wedge_{v_1, v_1 \in \Delta} \neg past\text{-}tape(\alpha, x, y, v_1, v_2) \\ (3) & error & \text{:-} \quad past\text{-}tape(\alpha, x, y, z, v), tape(\beta, x', y', z', v'), \wedge_{v_1, v_1 \in \Delta} \neg tape(\beta, x, y, v_1, v_2)\end{array}$$

The next rule ensures that the timestamp β used in the input *tape* is the successor of the maximum timestamp α of configurations already recorded in *past-tape*, if such a successor exists. We denote by $\varphi_{next}(\alpha, \beta)$ the following conjunction of literals, which says that α is the maximum configuration timestamp in *past-tape* and β is its successor index:

$$past\text{-}tape(\alpha, x, y, z, v), past\text{-}tape(\alpha', \alpha, \beta, z', v'), \wedge_{v_1, v_2 \in \Delta} \neg past\text{-}tape(\beta, 0, 1, v_1, v_2)$$

The rule is as follows:

$$(4) \quad error \quad \text{:-} \quad \varphi_{next}(\alpha, \beta), \wedge_{v_1, v_2 \in \Delta} \neg tape(\beta, 0, 1, v_1, v_2)$$

For the case when the maximum current timestamp has no successor index (i.e. we have run out of indexes), we need two additional rules to ensure that nothing can be inserted in *tape* without an error. Note that these rules are redundant if a successor index is available:

$$\begin{array}{lll}(5) & error & \text{:-} \quad tape(\beta, x, y, z, v), past\text{-}tape(\beta, x', y', z', v') \\ (6) & error & \text{:-} \quad tape(\beta, x, y, z, v), \neg past\text{-}index(\beta)\end{array}$$

Recall that *past-index* was constructed in Stage (1).

Finally, the next rules ensure that the new configuration input in *tape* is obtained from the previous one by a valid move of M. The move of M to be simulated is indicated by a unary input relation *move*. Suppose the move instructions of M are numbered $\{1, 2, \ldots, k\}$. The following rules ensure that *move* contains exactly one value among $\{1, 2, \ldots, k\}$:

$$\begin{array}{lll}(7) & error & \text{:-} \quad move(x), move(x'), x \neq x' \\ (8) & error & \text{:-} \quad \wedge_{i=1}^{k} \neg move(i)\end{array}$$

The next rules enforce that the new configuration is the result of a specified move from the previous configuration. For example, suppose the current state is q, the current cell is 0 and move i of M says that in state q and reading 0 M overwrites 0 by 1, its head moves to the right, and the new state is r. The following rule ensures that all cells of the previous configuration remain unchanged except the current one and the one to its right:

$$\begin{array}{lll}(9) & error & \text{:-} \quad move(i), \varphi_{next}(\alpha, \beta), past\text{-}tape(\alpha, x_0, x_1, z_1, 0), past\text{-}tape(\alpha, x_1, x_2, z_2, 0), \\ & & \quad\;\; \neg tape(\beta, x_1, x_2, z_2, 0) \\ (10) & error & \text{:-} \quad move(i), \varphi_{next}(\alpha, \beta), past\text{-}tape(\alpha, 0, 1, z, 0), \neg tape(\beta, 0, 1, z, 0)\end{array}$$

The last rules ensure that the current cell and the one to its right change according to the move:

$$\begin{array}{lll}(11) & error & \text{:-} \quad move(i), \varphi_{next}(\alpha, \beta), past\text{-}tape(\alpha, x_1, x_2, 0, q), \neg tape(\beta, x_1, x_2, 1, 0) \\ (12) & error & \text{:-} \quad move(i), \varphi_{next}(\alpha, \beta), past\text{-}tape(\alpha, x_1, x_2, 0, q), past\text{-}tape(\alpha, x_2, x_3, z, 0), \\ & & \quad\;\; \neg tape(\beta, x_2, x_3, z, r)\end{array}$$

Rules (11-12) work when the head of M is not at the rightmost end of the available tape, so the move to the right can be simulated. Observe that if this were not the case then there would also be no index available for use as a new configuration timestamp, since the number of configurations needed to reach the right end of the tape is at least the length of the tape. In this case rules (5-6) ensure that an error is output.

Other moves of M are simulated using similar rules. The transition to stage (3) occurs when a halting state h of M is reached (the rules implementing the transition are omitted).

Stage 3 At this last stage, the transducer outputs the word generated on the tape of M once the halting state (say, h) is reached. This is driven by inputting one at a time the indexes of the cells holding the word and outputting the symbol in the cell. Of course, the indexes have to be input in the right order. We use an input relation *cell*. The following error rules ensure that the sequence of indexes input into relation *cell* starts with 0 and proceeds in the right order:

$$
\begin{array}{lll}
\mathit{error} & \text{:-} & \mathit{cell}(\beta), \mathit{cell}(\beta'), \beta \neq \beta' \\
\mathit{error} & \text{:-} & \neg \mathit{cell}(0), \neg \mathit{past\text{-}cell}(0) \\
\mathit{error} & \text{:-} & \mathit{cell}(\beta), \mathit{past\text{-}cell}(\beta) \\
\mathit{error} & \text{:-} & \mathit{past\text{-}cell}(\alpha), \mathit{past\text{-}tape}(\alpha', \alpha, \beta, z, v), \neg \mathit{past\text{-}cell}(\beta), \neg \mathit{cell}(\beta)
\end{array}
$$

The last rules output the proposition p_z corresponding to each alphabet symbol z encountered in the cells:

$$
\begin{array}{l}
p_z \text{:- } \mathit{cell}(0), \mathit{past\text{-}tape}(\alpha, 0, 1, z, h), z \neq b \\
p_z \text{:- } \mathit{cell}(\beta), \mathit{past\text{-}tape}(\alpha, 0, 1, y, h), \mathit{past\text{-}tape}(\alpha, \beta, y, z, 0), z \neq b
\end{array}
$$

This completes Stage 3 and the construction of T. □

4.2 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 4.3 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 undecidability follows easily from the proof of Theorem 4.2. We use the undecidability of whether $\epsilon \in L$ where L is r.e. Let M be a nondeterministic Turing machine generating precisely the words in L starting from the empty tape. Let T be the Spocus transducer constructed from M in the proof of Theorem 4.2, which generates the prefix closure of L. Consider the $\mathcal{T}_{sdi}$ sentence:

$$\psi \equiv \forall\alpha\forall x[\mathit{past\text{-}tape}(\alpha, 0, 1, x, h) \ \rightarrow \ x \neq b]$$

where h is the halting state of M and b is the blank symbol. Then $\epsilon \notin L$ iff every error-free run of T satisfies ψ. Thus, the latter question is undecidable. □

We next show the decidability result for the case when *error* is defined by a set of rules where negation is not used on state literals.

Theorem 4.4 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 (Σ_2^p if the schema of 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 we have to consider only "short" runs of the Spocus transducer. More precisely, let $\psi \in \mathcal{T}_{sdi}$. As in the proof of Theorem 4.1, we can assume without loss of generality that ψ is of the form

$$\forall \vec{x}[A_1 \wedge \ldots \wedge A_m \rightarrow (L_1 \vee \ldots \vee L_n)]$$

where $A_i, 1 \leq i \leq m$ are literals over **state**, **db**, **in** with all variables in $\vec{x}$ occurring in positive literals, and $L_i, 1 \leq i \leq n$ are positive literals. Satisfaction of ψ at every stage in every error-free run of T is equivalent to unsatisfiability of the sentence

$$\theta \equiv \exists \vec{x}[A_1 \wedge \ldots \wedge A_m \wedge \neg L_1 \ldots \wedge \neg L_n].$$

in error-free runs of T. Therefore, it is sufficient to show that satisfiability of such sentences in error-free runs is decidable. Let k be the number of positive state literals among the A_i. We show the following:

(†) if there exists some finite error-free run of T such that θ is satisfied at the end of the run then there exists some error-free run of T of length $k+1$ such that θ is satisfied at the end of that run.

(‡) Satisfiability of θ on error-free runs of length $k+1$ can be reduced to satisfiability of an $\exists^*\forall^*$FO sentence over a signature consisting of **db**, and $k+1$ copies of **in**.

Clearly, proving (†) and (‡) is sufficient to establish the statement of the theorem. We first show (†). Suppose there exists an error-free run of T with input sequence $I_1 \ldots I_h$ such that θ is satisfied at step h. Let $\vec{a}$ be such that

$$[A_1 \wedge \ldots \wedge A_m \wedge \neg L_1 \ldots \wedge \neg L_n]$$

is satisfied at stage h with $\vec{x} = \vec{a}$. Let $A_i(\vec{a})$ and $L_j(\vec{a})$ be the literals A_i and L_j instantiated with $\vec{a}$. Thus, the quantifier-free sentence

$$[A_1(\vec{a}) \wedge \ldots \wedge A_m(\vec{a}) \wedge \neg L_1(\vec{a}) \ldots \wedge \neg L_n(\vec{a})]$$

is satisfied at stage h. Let $A_{i_1}, \ldots, A_{i_k}$ be the k positive state literals among the A_i. For each $p, 1 \leq p \leq k$, and literal $A_{i_p}(\vec{a}) = \textit{past-}R(\vec{a})$ there must exist an input I_{i_p} containing $R(\vec{a})$. We can assume without loss of generality that $i_1 \leq \ldots \leq i_k$. Now consider the input sequence $I_{i_1}, \ldots, I_{i_k}, I_h$. It is easy to see that the run of T on this input sequence remains error free. Also, if the length of the run is less than $(k+1)$ (because some of the i_p are the same) the run can be extended to an error-free run of length $(k+1)$ by simply keeping as many additional inputs as needed from the original sequence. Clearly,

$$[A_1(\vec{a}) \wedge \ldots \wedge A_m(\vec{a}) \wedge \neg L_1(\vec{a}) \ldots \wedge \neg L_n(\vec{a})]$$

continues to be satisfied at the last stage of the run on this input. It follows that

$$\exists \vec{x}[A_1 \wedge \ldots \wedge A_m \wedge \neg L_1 \ldots \wedge \neg L_n]$$

is also verified, so θ holds at the last stage of an error-free run of length $(k+1)$. This proves (†).

Next, consider (‡). Consider the signature consisting of **db** together with $(k+1)$ copies $\mathbf{in}_1$,...,$\mathbf{in}_{k+1}$ of **in**. Specifically, each $\mathbf{in}_j$ consists of one relation R_j for each relation R in **in**, of the same arity as R. The sentence whose satisfaction we must check has to take into account θ as well as the error-generating rules of T. For simplicity, assume T has a single error-generating rule

$$error \text{ :- } E_1, \ldots, E_q.$$

(The extension to several error rules is immediate.) The sentence is

$$\psi_\theta \wedge \psi_{error}$$

where ψ_θ and ψ_{error} are as follows. First,

$$\psi_\theta \;\equiv\; \exists \vec{x}[A'_1 \wedge \ldots A'_m \wedge L'_1 \wedge \ldots \wedge L'_n]$$

where the literals A'_i and L'_j are obtained from A_i and $\neg L_j$ as follows:

- a literal over **db** remains unchanged;
- a literal $(\neg)R(\vec{u})$ for $R \in \mathbf{in}$ is replaced by $(\neg)R_{k+1}(\vec{u})$;
- a literal *past*-$R(\vec{u})$ is replaced by $R_1(\vec{u}) \vee \ldots \vee R_k(\vec{u})$;
- a literal $\neg$*past*-$R(\vec{u})$ is replaced with $\neg R_1(\vec{u}) \wedge \ldots \wedge \neg R_k(\vec{u})$.

The sentence ψ_{error} equals $\psi_1 \wedge \ldots \wedge \psi_{k+1}$ where each ψ_j states that *error* is not generated at step j of the run. More precisely,

$$\psi_j \equiv \forall \vec{y}[E'_1 \vee \ldots \vee E'_q]$$

where each E'_i is obtained by essentially negating E_i as follows:

- a literal E_i over **db** is replaced by its negation;
- a literal E_i of the form $R(\vec{u})$ for $R \in \mathbf{in}$ is replaced by $\neg R_j(\vec{u})$ and $\neg R(\vec{u})$ is replaced by $R_j(\vec{u})$;
- a literal E_i of the form *past*-$R(\vec{u})$ is replaced with $\wedge_{i<j} \neg R_i(\vec{u})$.

Altogether, the sentence $\psi_\theta \wedge \psi_{error}$ is an $\exists^*\forall^*$FO sentence over the extended signature. The complexity of checking its satisfiability is NEXPTIME in the sentence, which is also NEXPTIME in T and θ. □

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 4.5 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: The argument is similar to the proof of Theorem 4.3 and uses the undecidability of whether $\epsilon \in L$ where L is an r.e. language. We make use again of the construction in Theorem 4.2. Let M be a nondeterministic Turing machine generating precisely the words in L starting from the empty tape. Let T be the Spocus transducer constructed from M in the proof of Theorem 4.2. Now consider a second Spocus transducer T_ϵ which is identical to T except for the additional error rule:

$$error \text{ :- } past\text{-}tape(\alpha, 0, 1, b, h)$$

(recall that h is the halting state of M and b is the blank tape symbol). Clearly, the new error rule is fired iff $\epsilon \in L$. It easily follows that $\epsilon \notin L$ iff every error-free run of T is also an error-free run of T_ϵ. □

The last result shows decidability of containment for an interesting special case, similar to the one in Theorem 4.4:

Theorem 4.6 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 (Σ_2^p if the schema of is fixed) whether every error-free run of T_1 is an error-free run of T_2.

Proof: The proof is very close to that of Theorem 4.4. Suppose there exists an error-free run of T_1 which is not an error-free run of T_2. In such a run, T_2 outputs *error* at some point in the run. Consider the first time this happens (so the prefix of the run up to that point generates no *error* in T_1 nor in T_2). Testing the existence of such a run amounts to testing the existence of a run which is error-free for T_1 and T_2 up to the last stage; at the last stage, T_2 generates an error but T_2 does not. By an argument similar to that in the proof of Theorem 4.4, we can show that if such a run exists then there exists such a run of length bounded by the number of state literals in a rule of T_2 generating an error at the last stage, plus one. Lastly, the problem is reduced, as in the proof of Theorem 4.4, to testing satisfiability of an $\exists^*\forall^*$FO sentence over a signature corresponding to runs of that length. □

5 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

[AD98] S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *Proc. ACM Symp. on Principles of Database Systems*, pp. 254-263, 1998.

[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 Verlag, Berlin, 1997.

[BS28] P. Bernays and M. Schönfinkel. 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.

[H+99] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative Workflows that Support Easy Modification and Dynamic Browsing. Int'l. Joint Conference on Work Activities Coordination and Collaboration (WACC), San Francisco, pp. 69-78, 1999.

[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 *Theoretical Computer Science*, 104:167–183, Springer-Verlag, Berlin, 1981.

[PV98] P. Picouet and V.Vianu. Semantics and expressiveness issues in active databases. *J. of Computer and System Sciences*, 57(3): 325-355, 1998.

[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*, vol. 4, Springer Verlag, Berlin, 1983.

[SR86] M. Stonebraker and L. Rowe. The design of Postgres. In *Proc. ACM SIGMOD Int'l. Conf. on the Management of Data*, pp. 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] Y. Yesha and N. Adam. Electronic Commerce: An Overview. In N. Adam and Y. Yesha, editors, *Electronic Commerce.* Lecture Notes in Computer Science, Springer-Verlag, 1996.