A Relational Framework for Information Extraction

Ronald Fagin
IBM Research – Almaden
San Jose, CA, USA

Benny Kimelfeld*
Technion
Haifa, Israel

Frederick Reiss
IBM Research – Almaden
San Jose, CA, USA

Stijn Vansummeren
Universite Libre de Bruxelles (ULB)
Bruxelles, Belgium

ABSTRACT

Information Extraction commonly refers to the task of populating a relational schema, having predefined underlying semantics, from textual content. This task is pervasive in contemporary computational challenges associated with Big Data. In this article we provide an overview of our work on *document spanners*—a relational framework for Information Extraction that is inspired by rule-based systems such as IBM's SystemT.

Categories and Subject Descriptors

F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Algebraic language theory, Classes defined by grammars or automata, Operations on languages*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata, Relations between models*; H.2.4 [**Database Management**]: Systems—*Textual databases* ; I.5.4 [**Pattern Recognition**]: Applications—*Text processing*

General Terms

Theory

Keywords

Information extraction, document spanners, regular expressions, automata, inconsistency, prioritized repairs

1. INTRODUCTION

Information Extraction (IE) refers to the task of discovering structured information in textual content. More precisely, the goal in IE is to populate a predefined relational schema that has predetermined underlying semantics, by correctly detecting the values of records in a given text document or a collection of text documents. Popular tasks in the space of IE include *named entity recognition* [29] (identify proper names in text, and classify those into a predefined set of categories such as *person* and *organization*), *relation extraction* [34] (extract tuples of entities that satisfy a predefined relationship, such as *person-organization*), *event extraction* [3] (find events of predefined types along with their key players, such as *nomination* and *nominee*), *temporal information extraction* [15, 25] (associate mentions of facts with mentions of their validity period, such as *nomination-date*), and *coreference resolution* [27] (match between phrases that refer to the same entity, such as "Obama," "the President," and "him").

As a discipline, IE began with the DARPA Message Understanding Conference (MUC) in 1987 [22]. While early work in the area focused largely on military applications, this task is nowadays pervasive in a plethora of computational challenges, in particular those associated with Big Data, such as social media analysis [6], machine data analysis [21], healthcare analysis [33], semantic search [35], and customer relationship management [2]. In a typical text-analytics pipeline (e.g., [32]), the output of IE is fed into a cleaning and/or fusion component, such as an entity-resolution algorithm, that in turn produces input for a global processing phase (e.g., statistical analysis or data mining). Contemporary business models like cloud computing, along with analytics platforms like Hadoop, facilitate such data analyses for a broad range of individuals and organizations.

Most information extraction systems incorporate a notion of *rules* in a domain-specific *rule language*. These rules may define the entire extraction task, or produce features for downstream statistical models. The rules may be manually coded, or automatically learned. The choice of a rule language comprises an important part of an IE system's design. Designing such a language involves navigating a number of tradeoffs, with the most important of these being that of simplicity versus expressivity. Keeping a rule language simple pays dividends in multiple ways. A simple rule language, with relatively few language constructs and a straightforward semantics, is easier for users to understand and debug, easier for learning algorithms to train, and easier for a rule engine to execute with high throughput. But such simplicity can easily compromise the expressiveness of

*Taub Fellow, supported by the Taub Foundation

the rule language, and thus lower the quality of extraction results. Limited expressiveness of a rule language may force developers to augment the rules with custom code in a general-purpose language like Java or Python. This practice, though often necessary to achieve acceptable accuracy, makes development, maintenance, and performance tuning significantly more difficult.

This article describes our recent work on a formal framework for examining the expressivity of IE rule languages. The framework, called *document spanners*, leverages known principles of database management. The framework itself is introduced in Sections 2 and 3. In Section 4 we give results on expressiveness, and in Section 5 we discuss conflict resolution within the framework. We discuss the impact of the string-equality operator in Section 6, and conclude in Section 7.

2. DOCUMENT SPANNERS

In this section, we give some preliminary definitions and recall the formalism of *document spanners* [19].

We fix a finite alphabet Σ of *symbols*. We denote by Σ^* the set of all finite strings over Σ, and by Σ^+ the set of all finite strings of length at least one over Σ. For clarity of context, we will often refer to a string in Σ^* as a *document*. [1] A *language over* Σ is a subset of Σ^*. Let $\mathbf{d} = \sigma_1 \cdots \sigma_n \in \Sigma^*$ be a document. The length n of $\mathbf{d}$ is denoted by $|\mathbf{d}|$. A *span* identifies a substring of $\mathbf{d}$ by specifying its bounding indices. Formally, a span of $\mathbf{d}$ has the form $[i, j\rangle$, where $1 \leqslant i \leqslant j \leqslant n+1$. If $[i, j\rangle$ is a span of $\mathbf{d}$, then $\mathbf{d}_{[i,j\rangle}$ denotes the substring $\sigma_i \cdots \sigma_{j-1}$. Note that $\mathbf{d}_{[i,i\rangle}$ is the empty string, and that $\mathbf{d}_{[1,n+1\rangle}$ is $\mathbf{d}$. The more standard notation would be $[i, j)$, but we use $[i, j\rangle$ to distinguish spans from intervals. For example, $[1, 1)$ and $[2, 2)$ are both the empty interval, hence equal, but in the case of spans we have $[i, j\rangle = [i', j'\rangle$ if and only if $i = i'$ and $j = j'$ (and in particular, $[1, 1\rangle \neq [2, 2\rangle$). We denote by $\mathsf{Spans}(\mathbf{d})$ the set of all the spans of $\mathbf{d}$. Two spans $[i, j\rangle$ and $[i', j'\rangle$ of $\mathbf{d}$ are *disjoint* if $j \leqslant i'$ or $j' \leqslant i$, and they *overlap* otherwise. Finally, $[i, j\rangle$ *contains* $[i', j'\rangle$ if $i \leqslant i' \leqslant j' \leqslant j$.

EXAMPLE 2.1. In all of the examples throughout the article, we consider the example alphabet Σ which consists of the lowercase and capital letters from the English alphabet (i.e., `a`,...,`z` and `A`,...,`Z`), the comma symbol ("`,`"), and the underscore symbol ("`_`") that stands for whitespace. (We use a restricted alphabet for simplicity.) Figure 1 depicts an example document $\mathbf{d}$ in Σ^*. For ease of later reference, it also depicts the index of each character in $\mathbf{d}$. Figure 2 shows two tables containing spans of $\mathbf{d}$. Observe that the spans in the left table of Figure 2 are those that correspond to words in $\mathbf{d}$ that are names of US states (Georgia, Washington and Virginia). For example, the span $[21, 28\rangle$ corresponds to Georgia. We will further discuss the meaning of these tables later. □

We fix an infinite set SVars of *(span) variables*; spans may be assigned to these variables. The sets Σ^* and SVars are disjoint. For a finite set $V \subseteq \mathsf{SVars}$ of variables and a document $\mathbf{d} \in \Sigma^*$, a $(V, \mathbf{d})$-*tuple* is a mapping $\mu\colon V \to \mathsf{Spans}(\mathbf{d})$ that assigns a span of $\mathbf{d}$ to each variable in V. A $(V, \mathbf{d})$-*relation* is a set of $(V, \mathbf{d})$-tuples. A *document spanner* (or just *spanner* for short) is a function P that is associated with a finite set V of variables, denoted $\mathsf{SVars}(P)$, and that maps every document $\mathbf{d}$ to a $(V, \mathbf{d})$-relation.

EXAMPLE 2.2. Throughout the article we will define several spanners. Two of those are denoted as $[\![\rho_{\mathsf{stt}}]\!]$ and $[\![\rho_{\mathsf{loc}}]\!]$, where $\mathsf{SVars}([\![\rho_{\mathsf{stt}}]\!]) = \{x\}$ and $\mathsf{SVars}([\![\rho_{\mathsf{loc}}]\!]) = \{x_1, x_2, y\}$. Later we will explain the meaning of the $[\![\cdot]\!]$ brackets , and specify what exactly each spanner extracts from a given document. For now, the span relations (tables) in Figure 2 show the results of applying the two spanners to the document $\mathbf{d}$ of Figure 1. □

Let P be a spanner with $\mathsf{SVars}(P) = V$. Let $\mathbf{d} \in \Sigma^*$ be a document, and let $\mu \in P(\mathbf{d})$ be a $(V, \mathbf{d})$-tuple. We say that μ is *hierarchical* if for all variables $x, y \in \mathsf{SVars}(P)$ one of the following holds: *(1)* the span $\mu(x)$ contains $\mu(y)$, *(2)* the span $\mu(y)$ contains $\mu(x)$, *or (3)* the spans $\mu(x)$ and $\mu(y)$ are disjoint. As an example, the reader can verify that all the tuples in Figure 2 are hierarchical. We say that P is *hierarchical* if μ is hierarchical for all $\mathbf{d} \in \Sigma^*$ and $\mu \in P(\mathbf{d})$. Observe that for two variables x and y of a hierarchical spanner, it may be the case that, over the same document, one tuple maps x to a subspan of y, another tuple maps y to a subspan of x, and a third tuple maps x and y to disjoint spans. Finally, we say that P is *Boolean* if $\mathsf{SVars}(P)$ is empty. Note that when P is Boolean, its application to a string $\mathbf{d}$ is either the empty set (*false*) or the singleton that consists of the empty tuple (*true*).

3. SPANNER REPRESENTATION

By a *spanner representation system* we refer collectively to any manner of specifying spanners through finite objects. In this section we recall several representation systems that we have proposed and studied in previous work [18, 19]: regex formulas, spanner algebra, basic extraction programs, and automata.

3.1 Regex Formulas

Regular expressions are one of the oldest types of information extraction rule languages. Many of the systems deployed in early MUC competitions used regular

[1] This is a text-only document without figures or tables.

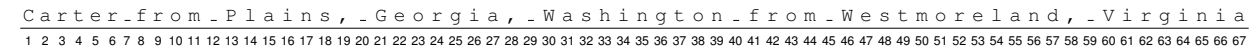

Figure 1: Document $\mathbf{d}$ in the running example

expressions over characters or token streams as their primary rule languages. A more recent example of a system with a regular expression-based rule language is the JAPE system [14], in which rules comprise regular expressions over streams of tokens, and rule evaluation is via a finite-state transducer.

A *regular expression with capture variables*, or just *variable regex* for short, is an expression in the following syntax that extends that of regular expressions:

$$\gamma \stackrel{\text{def}}{=} \varnothing \mid \epsilon \mid \sigma \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^* \mid x\{\gamma\} \quad (1)$$

The added alternative is $x\{\gamma\}$, where $x \in \mathsf{SVars}$. We denote by $\mathsf{SVars}(\gamma)$ the set of variables that occur in γ. We use γ^+ as abbreviations of $\gamma \cdot \gamma^*$.

A variable regex can be matched against a document in multiple ways, or more formally, there can be multiple parse trees showing that a document matches a variable regex. Each parse tree associates variables with spans. It is possible, however, that in a parse tree a variable is not associated with any span, or is associated with multiple spans. If every variable is associated with precisely one span, then the parse tree is said to be *functional*. A variable regex is called a *regex formula* if it has only functional parse trees on every input document. An example of a variable regex that is *not* a regex formula is $(x\{\texttt{a}\})^*$, because a match against `aa` assigns x to two spans. We refer to Fagin et al. [19] for the full formal definition of regex formulas. By RGX we denote the class of regex formulas. A regex formula γ is naturally viewed as representing a spanner, and by $[\![\gamma]\!]$ we denote the spanner that is represented by γ. Following are examples of spanners represented as regex formulas.

EXAMPLE 3.1. In the regex formulas of our running examples we will use the following conventions.

- `[a-z]` denotes the disjunction $\texttt{a} \vee \cdots \vee \texttt{z}$;
- `[A-Z]` denotes the disjunction $\texttt{A} \vee \cdots \vee \texttt{Z}$;
- `[a-zA-Z]` denotes `[a-z]` $\vee$ `[A-Z]`;
- Σ, by abuse of notation, denotes the regex formula recognizing all symbols in Σ, i.e., Σ denotes the disjunction `[a-zA-Z]` $\vee$ `,` $\vee$ `_`.

We now define several regex formulas that we will use throughout the article.

The following regex formula extracts tokens (which for our purposes now are simply complete words) from text. (Note that this is a simplistic extraction for the sake of presentation.)

$$\gamma_{\mathsf{tkn}} \stackrel{\text{def}}{=} \big(\epsilon \vee (\Sigma^* \cdot \texttt{\_})\big) \cdot x\{[\texttt{a-zA-Z}]^+\} \cdot \Big(\big((\texttt{,} \vee \texttt{\_}) \cdot \Sigma^*\big) \vee \epsilon\Big)$$

When applied to the document $\mathbf{d}$ of Figure 1, the resulting spans include $[1, 7\rangle$, $[8, 12\rangle$, $[13, 19\rangle$ and so on.

The following regex formula extracts spans that begin with a capital letter.

$$\gamma_{\mathsf{1cap}} \stackrel{\text{def}}{=} \Sigma^* \cdot x\{\texttt{[A-Z]} \cdot \Sigma^*\} \cdot \Sigma^*$$

When applied to the document $\mathbf{d}$ of Figure 1, the resulting spans include $[1, 7\rangle$, $[1, 3\rangle$, $[13, 19\rangle$, and so on.

The following regex formula extracts all the spans that span names of US states. For simplicity, we include just the three in Figure 1. For readability, we omit the concatenation symbol $\cdot$ between two alphabet symbols.

$$\gamma_{\mathsf{stt}} \stackrel{\text{def}}{=} \Sigma^* \cdot x\{\texttt{Georgia} \vee \texttt{Virginia} \vee \texttt{Washington}\} \cdot \Sigma^*$$

When applied to the document $\mathbf{d}$ of Figure 1, the resulting spans are $[21, 28\rangle$, $[30, 40\rangle$, and $[60, 68\rangle$.

The following regex formula extracts all the triples (x_1, x_2, y) of spans such that the string "`,_`" separates x_1 and x_2, and y is the span that starts where x_1 starts and ends where x_2 ends.

$$\gamma_{\texttt{,\_}} \stackrel{\text{def}}{=} \Sigma^* \cdot y\big\{x_1\{\Sigma^*\} \cdot \texttt{,\_} \cdot x_2\{\Sigma^*\}\big\} \cdot \Sigma^*$$

Let $\mathbf{d}$ be the document of Figure 1, and let V be the set $\{x_1, x_2, y\}$ of variables. The $(V, \mathbf{d})$-tuples that are obtained by applying $\gamma_{\texttt{,\_}}$ to $\mathbf{d}$ map (x_1, x_2, y) to triples like $([13, 19\rangle, [21, 28\rangle, [13, 28\rangle)$, and in addition, triples that do not necessarily consist of full tokens, such as the triple $([9, 19\rangle, [21, 23\rangle, [9, 23\rangle)$. □

3.2 Algebra over Spanners

Some IE systems use rule languages whose semantics derive from the relational calculus. For example, the Xlog system [28] system has a Datalog-based rule language, while SystemT [10] has a rule language based on SQL. These systems use rule engines that combine the relational algebra with automata for evaluating character-level primitives such as regular expressions. Such an algebraic runtime allows for efficient rule execution via query optimization. We can model this class of execu-

$[\![\rho_{\mathsf{stt}}]\!](\mathbf{d})$	
	x
μ_1	$[21, 28\rangle$
μ_2	$[30, 40\rangle$
μ_3	$[60, 68\rangle$

$[\![\rho_{\mathsf{loc}}]\!](\mathbf{d})$			
	x_1	x_2	y
μ_5	$[13, 19\rangle$	$[21, 28\rangle$	$[13, 28\rangle$
μ_4	$[21, 28\rangle$	$[30, 40\rangle$	$[21, 40\rangle$
μ_6	$[46, 58\rangle$	$[60, 68\rangle$	$[46, 68\rangle$

Figure 2: Results of spanners in the running example

tion environment by extending regex formulas with a relational algebra.

Let $\mathcal{R}$ be a representation system for spanners. Given a collection O of relational algebraic operators, we denote by $\mathcal{R}^O$ the closure of $\mathcal{R}$ under the operators of O. Here relational operators are extended pointwise to spanners. For example, consider $O = \{\bowtie\}$, where $\bowtie$ is the natural join operator. Then $\mathcal{R}^O$ consists of all spanners in $\mathcal{R}$, along with, for all spanners P_1 and P_2 definable in $\mathcal{R}$, the spanner $[\![P_1 \bowtie P_2]\!]$, which is defined by $[\![P_1 \bowtie P_2]\!](\mathbf{d}) = P_1(\mathbf{d}) \bowtie P_2(\mathbf{d})$ for all documents $\mathbf{d}$. Note in particular that the natural join here is based on span equality, not on string equality, since our relations contain spans.

We consider here three operators of positive relational algebra: union ($\cup$), projection (π), and natural join ($\bowtie$). Observe that the projection operator is parameterized by a sequence of variables from it operand spanner; that is, the operator has the form $\pi_{\mathbf{x}}$ where $\mathbf{x}$ is a sequence of variables. The standard typing rules for union and projection apply: union can only be applied to spanners P_1 and P_2 if $\mathsf{SVars}(P_1) = \mathsf{SVars}(P_2)$, and $\pi_{\mathbf{x}}$ is only applicable to spanner P if every member of $\mathbf{x}$ is in $\mathsf{SVars}(P)$. As usual, by $[\![\rho]\!]$ we denote the spanner that is represented by the algebraic expression ρ.

In the next example, we use the following notation. Let ρ be an expression in an algebra over RGX and let $\mathbf{x} = x_1, \ldots, x_n$ be a sequence of n distinct variables containing all the variables in $\mathsf{SVars}(\rho)$ (and possibly additional variables). Let $\mathbf{y} = y_1, \ldots, y_n$ be a sequence of distinct variables of the same length as $\mathbf{x}$. We denote by $\rho[\mathbf{y}/\mathbf{x}]$ the expression ρ' that is obtained from ρ by replacing every occurrence of x_i with y_i. If $\mathbf{x}$ is clear from the context, then we may write just $\rho[\mathbf{y}]$.

EXAMPLE 3.2. Using the regex formulas from Example 3.1, we define several $\mathsf{RGX}^{\{\cup,\pi,\bowtie\}}$-spanners.

- The spanner $\rho_{\mathsf{stt}} \stackrel{\text{def}}{=} \gamma_{\mathsf{tkn}} \bowtie \gamma_{\mathsf{stt}}$ extracts all the tokens that are names of US states. Note that, since $\mathsf{SVars}(\gamma_{\mathsf{tkn}}) = \mathsf{SVars}(\gamma_{\mathsf{stt}}) = \{x\}$, the natural join actually computes an intersection.
- The spanner $\rho_{\mathsf{1cap}} \stackrel{\text{def}}{=} \gamma_{\mathsf{tkn}} \bowtie \gamma_{\mathsf{1cap}}$ extracts all the tokens beginning with a capital letter.
- The spanner $\rho_{\mathsf{loc}} \stackrel{\text{def}}{=} \rho_{\mathsf{1cap}}[x_1/x] \bowtie \rho_{\mathsf{stt}}[x_2/x] \bowtie \gamma_{,\text{-}}$ extracts spans of strings including "*city*, *state*."

The results of applying the spanners $[\![\rho_{\mathsf{stt}}]\!]$ and $[\![\rho_{\mathsf{loc}}]\!]$ to the document $\mathbf{d}$ of Figure 1 are in Figure 2. Note that the right column of the right table in the figure is obtained through the spanner $\pi_y(\rho_{\mathsf{loc}})$, and the union of the left and middle columns is obtained through the spanner $\big(\pi_{x_1}(\rho_{\mathsf{loc}})\big) \cup \big(\pi_{x_2}(\rho_{\mathsf{loc}})\big)$. □

Later on, we will discuss several additional operators, including *selection*, *difference* and *complement*.

Loc	
f_1	$[13, 28\rangle$
f_2	$[21, 40\rangle$
f_3	$[46, 68\rangle$

Per	
f_4	$[1, 7\rangle$
f_5	$[13, 19\rangle$
f_6	$[21, 28\rangle$
f_7	$[30, 40\rangle$
f_8	$[46, 58\rangle$
f_9	$[60, 68\rangle$

PerLoc		
f_{10}	$[1, 7\rangle$	$[13, 28\rangle$
f_{11}	$[1, 7\rangle$	$[46, 68\rangle$
f_{12}	$[30, 40\rangle$	$[46, 68\rangle$

Figure 3: A $\mathbf{d}$-instance I over the signature of the running example

3.3 Basic Extraction Programs

In [18], we used the Datalog syntax for specifying spanners. We describe a basic form of this syntax (which we later extend) in this section.

A *signature* is a finite sequence $\mathbf{S} = \langle R_1, \ldots, R_m \rangle$ of distinct *relation symbols*, where each R_i has an arity $a_i > 0$. In this work, the *data* is a document $\mathbf{d}$, and entries in the instances of a signature are spans of $\mathbf{d}$. Formally, for a signature $\mathbf{S} = \langle R_1, \ldots, R_m \rangle$ and a document $\mathbf{d} \in \Sigma^*$, a $\mathbf{d}$-*instance* (*over* $\mathbf{S}$) is a sequence $\langle r_1, \ldots, r_m \rangle$, where each r_i is a relation of arity a_i over $\mathsf{Spans}(\mathbf{d})$; that is, r_i is a subset of $\mathsf{Spans}(\mathbf{d})^{a_i}$. A $\mathbf{d}$-*fact* (*over* $\mathbf{S}$) is an expression of the form $R(s_1, \ldots, s_a)$, where R is a relation symbol of $\mathbf{S}$ with arity a, and each s_i is a span of $\mathbf{d}$. If f is a $\mathbf{d}$-fact $R(s_1, \ldots, s_a)$ and I is a $\mathbf{d}$-instance, both over the signature $\mathbf{S}$, then we say that f is *a fact of* I if $(s_1, \ldots, s_a)$ is a tuple in the relation of I that corresponds to R. For convenience of notation, we identify a $\mathbf{d}$-instance with the set of its facts.

EXAMPLE 3.3. The signature $\mathbf{S}$ for our running example consists of three relation symbols:

- The unary relation symbol Loc stands for *location*;
- The unary relation symbol Per stands for *person*;
- The binary relation symbol PerLoc associates persons with locations.

We continue with our running example. Figure 3 shows a $\mathbf{d}$-instance over $\mathbf{S}$, where $\mathbf{d}$ is the document of Figure 1. This instance has 12 facts, and for later reference we denote them by $f_1, \ldots, f_{12}$. Note that there are quite a few mistakes in the table (e.g., the annotation of `Virginia` as a person by fact f_9); in the next section we will show how these are dealt with in the framework of this article. □

Let $\mathcal{R}$ be a spanner representation system. A *basic extraction program in* $\mathcal{R}$, or just *basic* $\mathcal{R}$*-program*, for short, is a triple $\langle \mathbf{S}, U, \varphi \rangle$, where $\mathbf{S}$ is a signature, U is a finite sequence $u_1, \ldots, u_m$ of *Horn rules*, and φ is an atomic formula over $\mathbf{S}$ (representing the result of the program). Here, an *atomic formula* φ is an expression of the form $R(x_1, \ldots, x_a)$, where R is an a-ary relation symbol in $\mathbf{S}$. A Horn rule has the form $R(y_1, \ldots, y_a) :-$

$\alpha_1 \wedge \cdots \wedge \alpha_k$, where R is a relation symbol of $\mathbf{S}$ of arity a, and each α_i is either an atomic formula over $\mathbf{S}$ or a spanner in $\mathcal{R}$. We make the requirement that each y_j occurs in at least one α_i. We denote by $\mathsf{BPR}\langle\mathcal{R}\rangle$ the class of basic $\mathcal{R}$-programs.

EXAMPLE 3.4. We now define a basic $\mathsf{RGX}^{\{\cup,\pi,\bowtie\}}$-program $\mathcal{E}$ in our running example. Intuitively, the goal of the program is to extract pairs (x, y), where x is a person and y is a location associated with x.[2] The signature is that of Example 3.3. The sequence U of rules is the following. Note that we are using the notation we established in the previous examples.

1. $\mathsf{Loc}(x) :- \rho_{\mathsf{loc}}[x]$ (see Example 3.2)
2. $\mathsf{Per}(y) :- \rho_{\mathsf{1cap}}[y]$ (see Example 3.2)
3. $\mathsf{PerLoc}(x, y) :- \mathsf{Per}(x) \wedge \mathsf{Loc}(y) \wedge \mathsf{precede}[x, y]$
4. RETURN $\mathsf{PerLoc}(x, y)$

In the above program, precede is the regex formula $\Sigma^* \cdot x\{\Sigma^*\} \cdot \Sigma^* \cdot y\{\Sigma^*\} \cdot \Sigma^*$. Hence, precede states that x terminates before y begins. □

3.4 Automata

Next, we recall a representation by means of automata. A *variable-set automaton* (or *vset-automaton*) is a tuple (Q, q_0, q_f, δ), where: Q is a finite set of *states*, $q_0 \in Q$ is an *initial state*, $q_f \in Q$ is an *accepting state*, and δ is a finite transition relation consisting of triples, each having one of the forms (q, σ, q'), (q, ϵ, q'), $(q, x\vdash, q')$ or $(q, \dashv x, q')$, where $q, q' \in Q$, $\sigma \in \Sigma$, and $x \in \mathsf{SVars}$. We denote by $\mathsf{SVars}(A)$ the set of variables that occur in the transitions of A.

Let $\mathbf{d} = \sigma_1 \cdots \sigma_n$ be a document. A *configuration* of a vset-automaton $A = (Q, q_0, q_f, \delta)$, when running on $\mathbf{d}$, is a tuple $c = (q, V, Y, i)$, where $q \in Q$ is the *current state*, $V \subseteq \mathsf{SVars}(A)$ is the set of *active variables*, $Y \subseteq \mathsf{SVars}(A)$ is the set of *available variables*, and i is an index in $\{1, \ldots, n + 1\}$. A *run* $\mathbf{c}$ of A on $\mathbf{d}$ is a sequence $c_0, \ldots, c_m$ of configurations, where $c_0 = (q_0, \varnothing, \mathsf{SVars}(A), 1)$, and for $j = 0, \ldots, m - 1$ one of the following holds for $c_j = (q_j, V_j, Y_j, i_j)$ and $c_{j+1} = (q_{j+1}, V_{j+1}, Y_{j+1}, i_{j+1})$:

1. $V_{j+1} = V_j$, $Y_{j+1} = Y_j$, and either (a) $i_{j+1} = i_j + 1$ and $(q_j, s_{i_j}, q_{j+1}) \in \delta$ (ordinary transition), or (b) $i_{j+1} = i_j$ and $(q_j, \epsilon, q_{j+1}) \in \delta$ (epsilon transition).
2. $i_{j+1} = i_j$ and for some $x \in \mathsf{SVars}(A)$, either (a) $x \in Y_j$, $V_{j+1} = V_j \cup \{x\}$, $Y_{j+1} = Y_j \backslash \{x\}$, and we have $(q_j, x\vdash, q_{j+1}) \in \delta$ (variable insert), or (b) $x \in V_j$, $V_{j+1} = V_j \backslash \{x\}$, $Y_{j+1} = Y_j$ and $(q_j, \dashv x, q_{j+1}) \in \delta$ (variable remove).

Note that in a run, each configuration (q, V, Y, i) is such that V and Y are disjoint. The run $\mathbf{c} = c_0, \ldots, c_m$ is *accepting* if $c_m = (q_f, \varnothing, \varnothing, n+1)$. We let $\mathsf{ARuns}(A, \mathbf{d})$ denote the set of all accepting runs of A on $\mathbf{d}$. If $\mathbf{c} \in \mathsf{ARuns}(A, \mathbf{d})$, then for each $x \in \mathsf{SVars}(A)$ the run $\mathbf{c}$ has a unique configuration $c_b = (q_b, V_b, Y_b, i_b)$ where x occurs in the current version of V (i.e., V_b) for the first time; and later than that $\mathbf{c}$ has a unique configuration $c_e = (q_e, V_e, Y_e, i_e)$ where x occurs in the current version of V (i.e., V_e) for the last time; the span $[i_b, i_e\rangle$ is denoted by $\mathbf{c}(x)$. By $\mu^{\mathbf{c}}$ we denote the $\mathbf{d}$-tuple that maps each variable $x \in \mathsf{SVars}(A)$ to the span $\mathbf{c}(x)$. The spanner $[\![A]\!]$ that is represented by A is the one where $\mathsf{SVars}([\![A]\!])$ is the set $\mathsf{SVars}(A)$, and where $[\![A]\!](\mathbf{d})$ is the $(\mathsf{SVars}(A), \mathbf{d})$-relation $\{\mu^{\mathbf{c}} \mid \mathbf{c} \in \mathsf{ARuns}(A, \mathbf{d})\}$. We denote by $\mathsf{VA}_{\mathsf{set}}$ the set of all variable-set automata.

As a simple example, Figure 4 depicts a vset-automaton that generates all tuples over $y_1, \ldots, y_m$

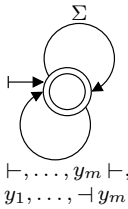

Figure 4: A vset-automaton that generates all tuples over $y_1, \ldots, y_m$

We remark that in [19] we have defined another type of automata for representing spanners, called *variable-stack* automata. We do not consider those in this article.

4. REGULAR SPANNERS AND EXPRESSIVENESS

We now give results on the expressiveness of the representation systems of the previous section. Given a representation system $\mathcal{R}$, we denote by $[\![\mathcal{R}]\!]$ the class of spanners definable by $\mathcal{R}$. The following theorem shows that several of the representation systems defined in the previous section have the same expressive power.

THEOREM 4.1. [18,19] *The following representation systems have precisely the same expressive power.*

- *The closure of regex formulas under union, projection and natural join.*
- *The basic* RGX*-programs.*
- *The vset-automata.*

That is, $[\![\mathsf{RGX}^{\{\cup,\pi,\bowtie\}}]\!] = [\![\mathsf{BPR}\langle\mathsf{RGX}\rangle]\!] = [\![\mathsf{VA}_{\mathsf{set}}]\!]$.

A spanner is *regular* if it is definable in the representation systems of Theorem 4.1. We denote by REG the set of expressions in $\mathsf{RGX}^{\{\cup,\pi,\bowtie\}}$. Hence, all of the representation systems of the theorem capture precisely

[2] In real life, such a program would of course be much more involved; here it is simplistic, for the sake of presentation.

$[\![\mathsf{REG}]\!]$. Note, however, that regex formulas are strictly less expressive than regular spanners. This is true, since a spanner defined by a regex formula is necessarily hierarchical. The following theorem shows that regex formulas capture *precisely* those regular spanners that are hierarchical.

THEOREM 4.2. [19] *A spanner P is definable in* RGX *if and only if P is both regular and hierarchical.*

Next, we discuss the selection operator. Let R be a k-ary string relation, and let P be a spanner. The string-selection operator ς^R is parameterized by k span variables $x_1, \ldots, x_k$ and may be written as $\varsigma^R_{x_1,\ldots,x_k}$. If P' is $\varsigma^R_{x_1,\ldots,x_k} P$, then $P'(\mathbf{d})$ is the restriction of $P(\mathbf{d})$ to those $\mathbf{d}$-tuples μ such that $(\mathbf{d}_{\mu(x_1)}, \ldots, \mathbf{d}_{\mu(x_k)}) \in R$. For example, if R is the binary relation consisting of all the pairs of strings that start with the same symbol, then $\varsigma^R_{x,y} P(\mathbf{d})$ is obtained from $P(\mathbf{d})$ by removing all the tuples γ in which the strings spanned by $\gamma(x)$ and $\gamma(y)$ start with different symbols.

A *string relation* is a relation over Σ^*. A k-ary string relation R is *recognizable* [7, 16] if it is a finite union of Cartesian products $L_1 \times \cdots \times L_k$, where each L_i is a regular language over Σ. We have the following.

THEOREM 4.3. [19] *Let R be a string relation.* RGX *is closed under the selection operator ς^R if and only if R is recognizable.*

Finally, we discuss difference and complementation. We denote by $\setminus$ the difference operator, and by $\sim$ the complement operator. Here, the *complement* of a spanner P is the spanner Q that has the same variables as P, and for every document $\mathbf{d}$, the tuples in $Q(\mathbf{d})$ are precisely those involving spans of $\mathbf{d}$ that are not in P. (Note that $Q(\mathbf{d})$ is finite since there only finitely many spans over $\mathbf{d}$.) Difference is defined as usual.

THEOREM 4.4. [19] *Regular spanners are closed under difference and complement; that is:*

$$\mathsf{REG} = \mathsf{REG}^{\{\setminus,\sim\}} = \mathsf{RGX}^{\{\cup,\pi,\bowtie,\setminus,\sim\}}$$

5. CONFLICT RESOLUTION

It is a common practice for different rules of an IE rule set to match the same region of text in different ways. Allowing this kind of overlap simplifies the task of developing and maintaining the rules if the rules are written by hand; and it simplifies the learning problem in systems that induce rules from examples. Nearly every IE rule system in use today allows for conflicting rules and provides language features for resolving these conflicts. Examples of such language features include the controls in the JAPE rule language [14] and the "consolidate" clause in SystemT's AQL [10]. The sections that follow describe our theoretical framework for a declarative language for specifying policies for conflict resolution [18].

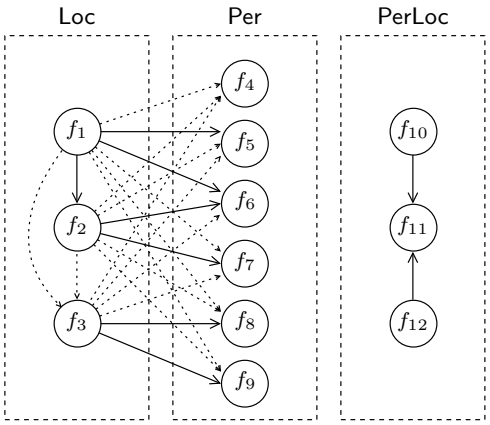

Figure 5: A conflict graph with priorities in the running example

5.1 Conflicts and Priorities

Observe that the instance of Figure 3 contains several conflicting facts. For example, f_2 represents a location, but it has a nonempty intersection with f_6 and f_7, which stand for person mentions. The database research community has established the concept of *repairs* as a mechanism for handling inconsistencies in a declarative fashion [5]. Conventionally, *denial constraints* are specified to declare sets of facts that cannot co-exist in a consistent instance. A *repair* of an inconsistent instance is a consistent subinstance that is not properly contained in any other consistent subinstance.

We adapt the concept of denial constraints to our setting. In the world of IE, the repairs are not necessarily all equal. In fact, in every example we are aware of, the developer has a clear preference as to which facts to exclude when a denial constraint is violated. Therefore, instead of the traditional repairs, we will use the notion of *prioritized repairs* of Staworko et al. [30], which extends repairing by incorporating priorities.

Let $\mathbf{S}$ be a signature, let $\mathbf{d}$ be a document, and let I be a $\mathbf{d}$-instance over $\mathbf{S}$. A *conflict hypergraph* for I is a hypergraph H over the facts of I; that is, $H = (V, E)$ where V is the set of I's facts and E is a collection of hyperedges (subsets of V). Intuitively, the hyperedges represent sets of facts that together are in conflict. A *priority relation* for I is a binary relation $\succ$ over the facts of I. If f and f' are facts of I, then $f \succ f'$ means intuitively that f is preferred to f'. A *repair* of I is a maximal subinstance of I that does not contain any hyperedge of H. To accommodate priorities in cleaning, we use the notion of *Pareto optimality* [30]: a consistent subinstance J is an *improvement* of a consistent subin-

Loc		Per		PerLoc	
f_1	$[13, 28\rangle$	f_4	$[1, 7\rangle$	f_{10}	$[1, 7\rangle$	$[13, 28\rangle$
f_3	$[46, 68\rangle$	f_7	$[30, 40\rangle$	f_{12}	$[30, 40\rangle$	$[46, 68\rangle$

Figure 6: A d-instance J_3 over the signature of the running example

stance J' if there is a fact $f \in J \backslash J'$ such that $f \succ f'$ for all $f' \in J' \backslash J$; an *optimal repair* is a consistent subinstance that has no improvement. It is easy to see that every optimal repair is also a repair in the sense of Arenas et al. [5].

EXAMPLE 5.1. Recall the instance I of our running example (Figure 3). Figure 5 shows both a conflict hypergraph (which is a graph in this case) and a priority relation over I. Specifically, the figure has two types of edges. Dotted edges (with small arrows) define priorities, where $f_i \rightarrow f_j$ denotes that $f_i \succ f_j$. Later, we shall explain the preferences (such as $f_1 \succ f_4$). Solid edges (with bigger arrows) define both conflicts and priorities: $f_i \rightarrow f_j$ denotes that $\{f_i, f_j\}$ is an edge of the conflict hypergraph, and that $f_i \succ f_j$.

Consider the following sets of facts.

$$J_1 \stackrel{\text{def}}{=} \{f_2, f_3, f_4, f_5, f_{11}\}$$
$$J_2 \stackrel{\text{def}}{=} (J_1 \cup \{f_1, f_7\}) \backslash \{f_2, f_5\}$$
$$J_3 \stackrel{\text{def}}{=} (J_2 \cup \{f_{10}, f_{12}\}) \backslash \{f_{11}\}$$

Observe that each J_i is consistent. J_2 is an improvement of J_1, since both $f_1 \succ f_2$ and $f_1 \succ f_5$ hold, and J_3 (depicted in Figure 6) is an improvement of J_2, since $f_{10} \succ f_{11}$ (and $f_{12} \succ f_{11}$). Note that J_3 is not an improvement of J_1, since no fact in J_3 is preferred to both f_2 and f_{11}. So "is an improvement of" is not transitive. The reader can verify that J_3 is an optimal repair, and in fact, the *unique* optimal repair. □

We note that another notion of optimality proposed by Staworko et al. [30] is *global optimality*, where J is an *improvement* of J' if $J \neq J'$ and for every fact $f' \in J' \backslash J$ there is a fact $f \in J \backslash J'$ such that $f \succ f'$. For the cases considered in this article, the two semantics coincide [18]. But in general, the two concepts are different. For example, in a traditional relational database with functional dependencies, optimal repair checking (i.e., given I and J, determine whether J is an optimal repair) is solvable in polynomial time in the Pareto semantics, but coNP-complete in the global semantics [17, 30].

5.2 Denial Constraints and Priority Generating Dependencies

We now discuss the syntactic declaration of conflicts and priorities. To specify a conflict hypergraph at the signature level (i.e., to specify the conflict hypergraph for every instance), we use the formalism of denial constraints. Let $\mathbf{S}$ be a signature, and let $\mathcal{R}$ be a spanner representation system. A *denial constraint* in $\mathcal{R}$ (*over* $\mathbf{S}$), or just $\mathcal{R}$*-dc* (or simply *dc*) for short, has the form

$$\forall \mathbf{x}[P \rightarrow \neg \Psi(\mathbf{x})]$$

where $\mathbf{x}$ is a sequence of variables in SVars, P is a spanner specified in $\mathcal{R}$ with all of its variables in $\mathbf{x}$, and Ψ is a conjunction of atomic formulas over $\mathbf{S}$. We usually omit the universal quantifier, and specify a dc simply by $P \rightarrow \neg \Psi(\mathbf{x})$. Semantically, $P \rightarrow \neg \Psi(\mathbf{x})$ is interpreted in the usual first-order-logic sense while viewing P as a predicate that contains all of the tuples in its output; that is, $P \rightarrow \neg \Psi(\mathbf{x})$ is satisfied in a document $\mathbf{d}$ if for every $(\mathbf{x}, \mathbf{d})$-tuple μ, if $P(\mathbf{d})$ contains the restriction of μ to $\mathsf{SVars}(P)$, then at least one of the conjuncts of Ψ must be false under μ.

EXAMPLE 5.2. We now define dcs in our running example. Recall that $\mathsf{precede}$ is a regex formula stating that x terminates before y begins. We denote by $\mathsf{disjoint}$ the regex formula $\mathsf{precede}[x, y] \vee \mathsf{precede}[y, x]$. We denote by $\mathsf{overlap}$ an expression in REG that represents the complement of $\mathsf{disjoint}$. Note that $\mathsf{overlap}$ is indeed expressible by a regular spanner, since regular spanners are closed under complement (Theorem 4.4). Finally, we denote by $\mathsf{overlap}_{\neq}$ an expression in REG that restricts the pairs in $\mathsf{overlap}$ to those (x, y) satisfying $x \neq y$ (i.e., x and y are not the same span). It is easy to verify that $\mathsf{overlap}_{\neq}$ indeed is expressible by a regular spanner.

The following dc, denoted d_{loc}, states that the spans of locations are disjoint.

$$d_{\mathsf{loc}} := \mathsf{overlap}_{\neq}[x, y] \rightarrow \neg\big(\mathsf{Loc}(x) \wedge \mathsf{Loc}(y)\big)$$

Similarly, the following dc, denoted d_{lp}, states that spans of locations are disjoint from spans of persons.

$$d_{\mathsf{lp}} := \mathsf{overlap}[x, y] \rightarrow \neg\big(\mathsf{Loc}(x) \wedge \mathsf{Per}(y)\big) \quad \square$$

To specify a priority relation $\succ$, we use what is called in [18] a *priority generating dependency*, or just *pgd* for short. Let $\mathbf{S}$ be a signature, and let $\mathcal{R}$ be a spanner representation system. A pgd in $\mathcal{R}$ (for $\mathbf{S}$) has the form $\forall \mathbf{x}[P \rightarrow (\varphi(\mathbf{x}) \succ \varphi'(\mathbf{x}))]$, where $\mathbf{x}$ is a sequence of variables in SVars, P is a spanner specified in $\mathcal{R}$ with all of its variables in $\mathbf{x}$, and φ and φ' are atomic formulas over $\mathbf{S}$. Again, we usually omit the universal quantifier and write just $P \rightarrow (\varphi(\mathbf{x}) \succ \varphi'(\mathbf{x}))$. And again, the semantics of $P \rightarrow (\varphi(\mathbf{x}) \succ \varphi'(\mathbf{x}))$ is the obvious one: for all $(\mathbf{x}, \mathbf{d})$ tuples μ, if $P(\mathbf{d})$ contains the restriction of μ to $\mathsf{SVars}(P)$, then $f \succ f'$ where f and f' are the facts that are obtained from $\varphi(\mathbf{x})$ and $\varphi'(\mathbf{x})$, respectively, by replacing every variable x with the span $\mu(x)$.

EXAMPLE 5.3. The following pgd, p_{loc}, states (using the expression $\rho[x, y]$, which is defined shortly) that for spans in the unary relation Loc, spans that start earlier are preferred, and moreover, when two spans begin together, the longer one is preferred.

$$p_{\mathsf{loc}} := \rho[x, y] \rightarrow \big(\mathsf{Loc}(x) \succ \mathsf{Loc}(y)\big)$$

Here, $\rho[x, y]$ is the following expression in REG.

$$\pi_{x,y}\Big(\big(\Sigma^* \cdot x\{z\{\epsilon\} \cdot \Sigma^*\} \cdot \Sigma^*\big) \bowtie \big(\Sigma^* \cdot z\{\epsilon\} \cdot \Sigma^+ \cdot y\{\Sigma^*\} \cdot \Sigma^*\big)\Big) \vee \big(\Sigma^* \cdot x\{y\{\Sigma^*\}\Sigma^+\} \cdot \Sigma^*\big)$$

Intuitively, the first disjunct says that x begins before y, because x begins with the empty span z, and y begins strictly after z begins. The second disjunct says that x and y begin together, but x ends strictly after y ends.

The following pgd, denoted p_{lp}, states that all the facts of Loc are preferred to all the facts of Per (e.g., because the extraction made for Loc is deemed more precise). We use the Boolean spanner true that is true on every document.

$$p_{\mathsf{lp}} := \mathsf{true} \rightarrow \big(\mathsf{Loc}(x) \succ \mathsf{Per}(y)\big) \quad \square$$

As we will discuss in Section 5.4, common resolution strategies translate into a dc and a pgd, such that the dc is binary, and the pgd defines priorities precisely on the facts that are in conflict. To refer to such a case conveniently, we write $P \rightarrow (\varphi(\mathbf{x}) \rhd \varphi'(\mathbf{x}))$ to jointly represent the dc $P \rightarrow \neg(\varphi(\mathbf{x}) \wedge \varphi'(\mathbf{x}))$ and the pgd $P \rightarrow (\varphi(\mathbf{x}) \succ \varphi'(\mathbf{x}))$. We call such a constraint a *denial pgd*.

EXAMPLE 5.4. We use $\mathsf{contains}_{\neq}[x, y]$ to denote a regex formula that produces all pairs (x, y) of spans where x strictly contains y. Let $\mathsf{enclose}[z, x, y]$ denote a specification in REG that produces all the triples (z, x, y), such that z begins where x begins and ends where y ends. For presentation's sake, we avoid the precise specification of these formulas.

The following denial pgd, denoted dp_{enc}, states that in the relation PerLoc, two facts are in conflict if the span that covers the two elements (person and location) of the first fact strictly contains that span that covers the two elements of the second; in that case, the shorter span is prioritized (since a shorter span indicates closer relationship between the person and the location).

$$\mathsf{enclose}[z, x, y] \bowtie \mathsf{enclose}[z', x', y'] \bowtie \mathsf{contains}_{\neq}[z', z] \rightarrow \mathsf{PerLoc}[x, y] \rhd \mathsf{PerLoc}[x', y'] \quad \square$$

EXAMPLE 5.5. Consider again the $\mathbf{d}$-instance I of Figure 3. The reader can verify that dcs d_{loc} and d_{lp} from Example 5.2, the pgds p_{loc} and p_{lp} in Example 5.3 and the denial pgd dp_{enc} of Example 5.4, together define the conflicts and priorities discussed in Example 5.1 (Figure 5). $\square$

Let $\mathcal{R}$ be a spanner representation system. An *extraction program in* $\mathcal{R}$, or just $\mathcal{R}$*-program* for short, is similar to a basic $\mathcal{R}$-program, except that we now allow for *cleaning rules* in addition to the Horn rules. A *cleaning rule* has the form form CLEAN$(\delta_1, \ldots, \delta_d)$, where each δ_i is a dc or a pgd in $\mathcal{R}$ (for convenience, we will also allow denial pgds).

In the program of the following example, we specify an extraction program $\langle \mathbf{S}, U, \varphi \rangle$ using only U (a sequence of rules) along with a special RETURN statement that specifies φ. We then assume that $\mathbf{S}$ consists of precisely the relation symbols that occur in the program.

EXAMPLE 5.6. We now define the REG-program $\mathcal{E}$ of our running example. Intuitively, the goal of the program is to extract pairs (x, y), where x is a person and y is a location associated with x.[3] The signature is, as usual, that of Example 3.3. The sequence U of rules is the following. Note that we are using the notation we established in the previous examples.

1. $\mathsf{Loc}(x) :- \rho_{\mathsf{loc}}[x]$ (Example 3.2)
2. $\mathsf{Per}(y) :- \rho_{\mathsf{1cap}}[y]$ (Example 3.2)
3. CLEAN$(d_{\mathsf{loc}}, d_{\mathsf{lp}}, p_{\mathsf{loc}}, p_{\mathsf{lp}})$ (Examples 5.2 and 5.3)
4. $\mathsf{PerLoc}(x, y) :- \mathsf{Per}(x) \wedge \mathsf{Loc}(y) \wedge \mathsf{precede}[x, y]$ (Example 5.2)
5. CLEAN(dp_{enc}) (Example 5.4)
6. RETURN $\mathsf{PerLoc}(x, y)$

Note that lines 1, 2 and 4 are Horn rules, whereas lines 3 and 5 are cleaning rules. $\square$

Let $\mathcal{E} = \langle \mathbf{S}, U, \varphi \rangle$ be an $\mathcal{R}$-program and let $\mathbf{d}$ be a document. Suppose that $U = \langle u_1, \ldots, u_m \rangle$. Let $\mathbf{I}_0$ be the singleton $\{I_\varnothing\}$, where $I_\varnothing$ is the empty instance over $\mathbf{S}$. For $i = 1, \ldots, m$, we denote by $\mathbf{I}_i$ the result of executing the rules $u_1, \ldots, u_i$ as we describe below. Since the cleaning operation can result in multiple instances (optimal repairs), each $\mathbf{I}_i$ is a *set* of $\mathbf{d}$-instances, rather than a single one. For $i > 0$ we define the following.

1. If u_i is the Horn rule $R(x_1, \ldots, x_a) :- \alpha_1 \wedge \cdots \wedge \alpha_k$, then $\mathbf{I}_i$ is obtained from $\mathbf{I}_{i-1}$ by adding to each $I \in \mathbf{I}_{i-1}$ all the facts (over R) that are obtained by evaluating the rule over I.

2. If u_i is the cleaning rule CLEAN$(\delta_1, \ldots, \delta_d)$, then $\mathbf{I}_i$ is obtained from $\mathbf{I}_{i-1}$ by replacing each $I \in \mathbf{I}_{i-1}$ with all the optimal repairs of I, as defined by the conflict hypergraph and priorities implied by all the δ_j.

[3] Again, our program is simplistic, for the sake of presentation.

Recall that a spanner is a function that maps a document into a $(V, \mathbf{d})$-relation (see Section 2). An extraction program acts similarly, except that a document is mapped into a set of $(V, \mathbf{d})$-relations (since it branches into multiple optimal repairs); these are all the possible resulting relations φ. For a more precise definition of the output of an extraction program, see [18]. In practice, the common case is where the extraction program produces precisely one $(V, \mathbf{d})$-relation, and then we will view the extraction program simply as a spanner.

EXAMPLE 5.7. Consider again the REG-program $\mathcal{E}$ of Example 5.6. We will now follow the steps of evaluating the program $\mathcal{E}$ on the document $\mathbf{d}$ of our running example (Figure 1). It turns out that, in this example, each $\mathbf{I}_i$ is a singleton, since every cleaning operation results in a unique optimal repair. Hence, we will treat the $\mathbf{I}_i$ as instances.

1. In $\mathbf{I}_1$, the relation Loc is as shown in Figure 3, and the other two relations are empty.
2. In $\mathbf{I}_2$, the relations Loc and Per are as shown in Figure 3, and PerLoc is empty.
3. In $\mathbf{I}_3$, the relations Loc and Per are as shown in Figure 6, and PerLoc is empty. The cleaning process is described throughout Sections 5.1 and 5.2.
4. In $\mathbf{I}_4$, the relations Loc and Per are as in $\mathbf{I}_3$, and PerLoc is as shown in Figure 3.
5. $\mathbf{I}_5$ is the instance shown in Figure 6.

The result $\mathcal{E}(\mathbf{d})$ is the $(\{x, y\}, \mathbf{d})$-relation that has two mappings: the first maps (x, y) to $([1, 7\rangle, [13, 28\rangle)$, and the second to $([30, 40\rangle, [46, 48\rangle)$. □

5.3 Cleaning in REG-Programs

We now discuss fundamental properties of extraction programs, where we focus on the class of REG-programs.

In the framework of prioritized repairs [30], the priority relation is assumed to be acyclic. We did not make such an assumption, and a pgd can indeed define a cyclic priority relation in a given program. We would like to be able to test whether acyclicity is guaranteed, but unfortunately, as the next theorem implies, no such algorithm exists for general pgds.

Let c be a cleaning rule. We say that c is *acyclic* if, for every document $\mathbf{d}$ and $\mathbf{d}$-instance I over $\mathbf{S}$, the priority relation implied by the pgds of c is acyclic.

THEOREM 5.8. [18] *Whether a pgd in* REG *is acyclic is co-recursively enumerable but not recursively enumerable. In particular, it is undecidable.*

Recall that a spanner maps a document $\mathbf{d}$ into a $(V, \mathbf{d})$-relation, for a set V of variables, while an extraction program maps $\mathbf{d}$ into a *set* of $(V, \mathbf{d})$-relations. The next property we discuss for extraction programs is that of *unambiguity*, which is the property of having a single possible world when the program is evaluated over any given document. Formally, we say that extraction program $\mathcal{E}$ is *unambiguous* if $\mathcal{E}(\mathbf{d})$ is a singleton $(V, \mathbf{d})$-relation for every document $\mathbf{d}$. We may view an unambiguous extraction program $\mathcal{E}$ simply as a specification of a spanner. The following theorem states that, unfortunately, in the presence of cleaning rules unambiguity cannot be verified for regular extraction programs.

THEOREM 5.9. [18] *Whether a* REG*-program is unambiguous is co-recursively enumerable but not recursively enumerable. In particular, it is undecidable.*

We now give a sufficient and decidable condition for unambiguity, in the case where acyclicity is guaranteed. Let I be a $\mathbf{d}$-instance over a signature $\mathbf{S}$. Let H and $\succ$ be a conflict hypergraph for I and a priority relation over I, respectively. We say that $(\succ, H)$ satisfies the *minimum property* if every hyperedge h of H contains a minimum element, that is, an element a such that $b \succ a$ for every member of h other than a. Let c be a cleaning rule. We say that c is *minimum generating* if, for every document $\mathbf{d}$ and $\mathbf{d}$-instance I over $\mathbf{S}$, for the priority relation $\succ$ and conflict hypergraph H implied by c we have that $(\succ, H)$ satisfies the minimum property. We note that for acyclic priority relations, the minimum property is less strict than the *totality* property that Staworko et al. [30] gave as a condition for unambiguity. We have the following theorem.

THEOREM 5.10. [18] *Let $\mathcal{E}$ be an $\mathcal{R}$-program for some spanner representation system $\mathcal{R}$. If every cleaning rule of $\mathcal{E}$ is acyclic and minimum generating, then $\mathcal{E}$ is unambiguous.*

In addition, we have shown that the property of being minimum generating is decidable for regular spanners.

THEOREM 5.11. [18] *Whether a given cleaning rule in* REG *is minimum generating is decidable.*

Unfortunately, the property of being acyclic is undecidable, as stated in Theorem 5.8, and so is the property of being *both* acyclic and minimum generating. Hence, as future research it is of interest to find decidable properties that imply these two properties.

Next, we address the question of whether cleaning rules increase the expressive power of extraction programs. Let $\mathcal{R}$ be a spanner representation system. A cleaning rule c defined in $\mathcal{R}$ is said to be *$\mathcal{R}$-disposable* if the following holds: for every $\mathcal{R}$-program $\mathcal{E}$ that has c as its single cleaning rule, there exists a basic (non-cleaning) $\mathcal{R}$-program that is equivalent to $\mathcal{E}$. Of course, if every cleaning rule of $\mathcal{E}$ is $\mathcal{R}$-disposable, then $\mathcal{E}$ is equivalent to a basic $\mathcal{R}$-program.

We say that a denial pgd p is *$\mathcal{R}$-disposable* if the cleaning rule that consists of only p is $\mathcal{R}$-disposable.

The following theorem implies that cleaning rules, and in fact a single acyclic denial pgd, increase the expressive power of regular extraction programs. Recall that a program that uses an acyclic denial pgd as its single cleaning rule is unambiguous (Theorem 5.10).

THEOREM 5.12. [18] *There exists an acyclic denial pgd in* REG *that is not* REG*-disposable.*

5.4 JAPE Controls

JAPE [14] is an instantiation of the Common Pattern Specification Language (CPSL) [4], a rule based framework for IE. A JAPE program (or "phase") can be viewed as an extraction program where all the relation symbols are unary. JAPE has several built-in cleaning strategies called "controls." Here, we will define these strategies in our own terminology—denial pgds.

JAPE provides four controls (in addition to the All control stating that no cleaning is to be applied). These translate to the following denial pgds. Here, R is assumed to be a unary relation in an extraction program. Under the Appelt control, $R(x) \rhd R(y)$ holds if *(1)* x and y overlap and x starts earlier than y, *or (2)* x and y start at the same position but x is longer than y. The same strategy is used is also provided by SystemT [10] (as a "consolidator"). The First control is similar to Appelt with "longer" replaced with "shorter." The Brill control is similar to Appelt, with the exclusion of option (2); that is, $R(x) \rhd R(y)$ holds if x and y overlap and x starts earlier than y. The Once control states that a single fact should remain in R (unless R is empty), which is the one that starts earliest, where a tie is broken by taking the one that ends earliest. Hence, $R(x) \rhd R(y)$ if and only if x is that remaining fact and $x \neq y$.

It is easy to show that each of the above denial pgds is acyclic, and can be expressed in REG. For example, the Appelt control is presented in Example 5.3 with R being the relation symbol Loc. While the JAPE controls can significantly simplify the programming of spanners, they do not add expressive power to regular programs, as the following theorem states.

THEOREM 5.13. [18] *Each of the denial pgds that correspond to the four JAPE controls is* REG*-disposable.*

5.5 Regular Spanners and POSIX

A regex formula γ defines a spanner by considering all possible ways that input document $\mathbf{d}$ can be matched by γ; that is, it considers all possible (functional) parse trees of γ on $\mathbf{d}$. Each such parse tree generates a new $(V, \mathbf{d})$-tuple, where $V = \mathsf{SVars}(\gamma)$, in the resulting span relation. In contrast, regular-expression pattern-matching facilities of common UNIX tools, such as `sed` and `awk`, or programming languages such as Perl, Python, and Java, do not construct all possible parse trees. Instead, they employ a *disambiguation policy* to construct only a single parse tree among the possible ones. As a result, a regex formula in these tools always yields a single $(V, \mathbf{d})$-tuple per matched input document $\mathbf{d}$ instead of multiple such tuples.[4]

In this section, we discuss the POSIX disambiguation policy [20, 23], a policy which is followed by all POSIX compliant tools such as `sed` and `awk`. Formalizations of this policy have been proposed by Vansummeren [31] and Okui and Suzuki [26], and multiple efficient algorithms for implementing the policy are known [12, 24, 26].

POSIX disambiguates as follows when matching a document $\mathbf{d}$ against regex formula γ.[5] A formal definition may be found in [26, 31]. If γ is one of $\varnothing$, ϵ, or $\sigma \in \Sigma$ then at most one parse tree exists; disambiguation is hence not necessary. If γ is a disjunction $\gamma_1 \vee \gamma_2$, then POSIX first tries to match $\mathbf{d}$ against γ_1 (recursively, using the POSIX disambiguation policy to construct a unique parse tree for this match). Only if this fails it tries to match against γ_2 (again, recursively). If, on the other hand, γ is a concatenation $\gamma_1 \cdot \gamma_2$ then POSIX first determines the longest prefix $\mathbf{d_1}$ of $\mathbf{d}$ that can be matched by γ_1 such that the corresponding suffix $\mathbf{d_2}$ of $\mathbf{d}$ can be matched by γ_2. Then, $\mathbf{d_1}$ (respectively, $\mathbf{d_2}$) is recursively matched under the POSIX disambiguation policy by γ_1 (respectively, γ_2) to construct a unique parse tree for γ. When γ is a Kleene closure δ*, there are two cases. If $\mathbf{d}$ is empty, then the entire pattern γ is taken to match $\mathbf{d}$ (irrespective of whether δ itself matches $\mathbf{d}$), and disambiguation is not necessary. If, on the other hand, $\mathbf{d}$ is nonempty, then POSIX expands γ to $\delta \cdot \delta$*. In line with the rule for concatenation, it hence first determines the longest prefix $\mathbf{d_1}$ of $\mathbf{d}$ that can be matched by δ such that the corresponding suffix $\mathbf{d_2}$ of $\mathbf{d}$ can be matched by δ*. Then, a unique parse tree for $\mathbf{d}$ against γ is constructed by matching $\mathbf{d_1}$ recursively against δ and $\mathbf{d_2}$ against δ*.

The following example illustrates the POSIX policy.

EXAMPLE 5.14. Consider $\gamma = x\{(0 \vee 01)\} \cdot y\{(1 \vee \epsilon)\}$ and $\mathbf{d} = 01$. Under the POSIX disambiguation policy, subexpression $x\{(0 \vee 01)\}$ will match as much of $\mathbf{d}$ as possible while still allowing the rest of the expression, namely $y\{(1 \vee \epsilon)\}$, to match the remainder of $\mathbf{d}$. As such, $x\{(0 \vee 01)\}$ will match $\mathbf{d}$ entirely, and $y\{(1 \vee \epsilon)\}$ will match the empty string. We hence bind x to the span $[1, 3\rangle$ and y to $[3, 3\rangle$.

[4] While our syntax $x\{\gamma\}$ for variable binding is not directly supported in these tools, it can be mimicked through the use of so-called *parenthesized expressions* and *submatch addressing*.

[5] For simplicity, we restrict ourselves here to the setting where the entire input is required to match γ. Our results naturally extend to the setting where partial matches of $\mathbf{d}$ against γ are sought.

As another example, when $\gamma = (x\{0\} \cdot y\{(1 \vee \epsilon)\}) \vee (x\{01\} \cdot y\{(1 \vee \epsilon)\})$ and $\mathbf{d} = 01$, we bind x to the span $[1, 2\rangle$ and y to the span $[2, 3\rangle$ under the POSIX disambiguation policy. $\square$

By $\mathsf{posix}[\gamma]$ we denote the spanner represented by the regex formula γ under the POSIX disambiguation policy; this is the spanner such that $\mathsf{posix}[\gamma](\mathbf{d})$ is empty if $\mathbf{d}$ cannot be matched by γ, and consists of the unique $(V, \mathbf{d})$-tuple resulting from matching $\mathbf{d}$ against γ under the POSIX disambiguation policy otherwise.

The following theorem shows that the POSIX policy can be expressed in our cleaning framework.

THEOREM 5.15. [18] *For all regex formulas γ there exists a* REG*-program $\mathcal{E}$ such that for every document $\mathbf{d}$,*

$$\mathcal{E}(\mathbf{d}) = \{\mathsf{posix}[\gamma](\mathbf{d})\}.$$

While proving Theorem 5.15, we have observed that every cleaning rule we used in $\mathcal{E}$ is REG-disposable. Moreover, since the spanner $\mathsf{posix}[\gamma]$ is hierarchical, it follows by Theorem 4.2 that $\mathsf{posix}[\gamma]$ is itself definable in RGX by a regex formula δ. We then conclude the following theorem about POSIX, which is of interest independently of our framework.

THEOREM 5.16. [18] *For every regex formula γ, the spanner* $\mathsf{posix}[\gamma]$ *is definable in* RGX.

6. STRING EQUALITY

In this section we discuss the enrichment of regular spanners with the binary string-selection operator, denoted $\varsigma^{=}$. Given a spanner P and two variables $x, y \in \mathsf{SVars}(P)$, the application of $\varsigma^{=}_{x,y}$ selects all the tuples μ in which $\mathbf{d}_{\mu(x)} = \mathbf{d}_{\mu(y)}$. A *core spanner* [19] is a spanner definable in the algebra $\mathsf{RGX}^{\{\cup,\pi,\bowtie,\varsigma^{=}\}}$. We denote this algebra by Core.

It follows immediately from known literature on finite-state automata that core spanners have a strictly greater expressive power than regular spanners; that is, every regular spanner is a core spanner, and there are core spanners that are not regular [19]. An example of a non-regular core spanner is the following spanner, extracting all the pairs of spans with equal strings.

$$\varsigma^{=}_{x,y}\left((\Sigma^* x\{\Sigma^*\}\Sigma^*) \times (\Sigma^* y\{\Sigma^*\}\Sigma^*)\right)$$

Recall from Theorem 4.4 that regular spanners are closed under difference. The following theorem states that this is no longer the case for core spanners.

THEOREM 6.1. [19] *Assume that the alphabet Σ contains at least two symbols. Core spanners are* not *closed under difference; that is,*

$$[\![\mathsf{RGX}^{\{\cup,\pi,\bowtie,\varsigma^{=}\}}]\!] \subsetneq [\![\mathsf{RGX}^{\{\cup,\pi,\bowtie,\varsigma^{=},\setminus\}}]\!].$$

Next, we discuss the proof of Theorem 6.1. As noted in [19], the authors originally believed that the way to prove Theorem 6.1 would be to show that core spanners cannot simulate string *inequality* (i.e., select the tuples μ in which $\mathbf{d}_{\mu(x)} \neq \mathbf{d}_{\mu(y)}$). However, surprisingly, it turned out that this argument is false.

PROPOSITION 6.2. [19] *Core spanners are closed under the string-inequality operator.*

As a part of the proof of Theorem 6.1, we established the following lemma, which is of independent interest.

LEMMA 6.3. *Every core spanner is definable by an expression of the form $\pi_V SP$, where P defines a regular spanner, $V \subseteq \mathsf{SVars}(P)$, and S is a sequence of selections $\varsigma^{=}_{x,y}$ for $x, y \in \mathsf{SVars}(P)$.*

The proof of Theorem 6.1 then completes as follows. An easy observation is that core spanners are closed under the substring-of selection operator (i.e., select the tuples μ in which $\mathbf{d}_{\mu(x)}$ is a substring of $\mathbf{d}_{\mu(y)}$). But using Lemma 6.3 we have proved the following theorem.

THEOREM 6.4. [19] *Assume that the alphabet Σ contains at least two symbols. Core spanners are not closed under the* not-a-substring-of *binary operator.*

We complete this section with the following theorem, which implies that disposability of the JAPE controls no longer holds in the case of core spanner.

THEOREM 6.5. [18] *None of the JAPE denial pgds is* Core*-disposable.*

7. CONCLUDING REMARKS

We conclude this paper by some observations relating spanners to other formalisms, and some open questions.

Many practical regular expression pattern matching engines (such as the ones found in `sed`, `awk`, Perl, Java and Python) support a feature called *backreferences*. Using this feature, variables can not only bind to a substring during matching, but can also be used to repeat a previously matched substring. Regular expressions that have this feature are called *extended regular expressions* (xregex for short) [1, 8, 9]. It is known that xregex can recognize non-regular languages, such as $\{\mathbf{ss} \mid \mathbf{s} \in \Sigma^*\}$. Note that this language can be recognized by a Boolean core spanner. In [19] we established that xregex can also recognize languages that are not recognizable by any Boolean core spanner. The reverse question, whether Boolean core spanners can recognize languages that are not recognizable by any xregex, is still open.

Various languages for querying semi-structured and graph databases are based on regular expressions. A simple form of such queries are the *regular path queries*

(*RPQs*) that are applied to directed graphs with labeled edges [11, 13]. An RPQ identifies node pairs connected by a path such that the word formed by the edge labels belongs to a specified regular language. A *conjunctive regular path query* (*CRPQ*) applies conjunction and existential quantification (over nodes) to RPQs [11], and has been the subject of much investigation.

Superficially speaking, spanners and CRPQs are inherently different concepts: spanners operate on *strings* while CRPQs operate on *graphs* (directed, edge-labeled graphs); and the variables in the spanner world represent *spans*, while those in the CRPQ world represent *nodes*. However, it is possible to adjust CRPQs to represent spanners [19]. In terms of the data model, a string can be viewed as a special case of a graph, namely a simple path. Formally, given a string $\mathbf{s} = \sigma_1 \cdots \sigma_n$, we denote by $p(\mathbf{d})$ the simple path $1 \to 2 \to \cdots \to n+1$ (with the natural numbers $1, \ldots, n+1$ as nodes), where for $i = 1, \ldots, n$ the label of the edge $i \to i+1$ is σ_i. For technical reasons, it is necessary to mark the begin node 1 and end node n in this simple path with the two loops $1 \to 1$ and $(n+1) \to (n+1)$, labeled with new labels $\triangleright$ and $\triangleleft$ (not in the alphabet Σ). On this so-called *marked path*, a CRPQ can define a spanner over a set of span variables V by introducing, for each $x \in V$, two CRPQ variables: one that will indicate the start position of the span matched by x and one that indicates the end position of the span matched by x. Using this representation of spanners through CRPQ, one can show that $[\![\mathsf{REG}]\!]$ is exactly captured by unions of CRPQs, while $[\![\mathsf{Core}]\!]$ is exactly captured by unions of CRPQs extended with string equality [19].

As we have seen in this article, we have identified several representation systems for regular and core spanners that are equivalent in expressive power. While our proofs of these equivalences describe effective translations between the representation systems, it would be interesting to study the inherent complexity of these translations in order to establish their relative succinctness. A second question that deserves further attention is the complexity of evaluating spanners expressed in the various representation systems.

8. REFERENCES

[1] A. V. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 255–300. North Holland, 1990.

[2] J. Ajmera, H.-I. Ahn, M. Nagarajan, A. Verma, D. Contractor, S. Dill, and M. Denesuk. A CRM system for social media: challenges and experiences. In *WWW*, pages 49–58, 2013.

[3] C. Aone and M. Ramos-Santacruz. Rees: A large-scale relation and event extraction system. In *ANLP*, pages 76–83, 2000.

[4] D. E. Appelt and B. Onyshkevych. The common pattern specification language. In *Proceedings of the TIPSTER Text Program: Phase III*, pages 23–30, Baltimore, Maryland, USA, 1998. Association for Computational Linguistics.

[5] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79. ACM, 1999.

[6] E. Benson, A. Haghighi, and R. Barzilay. Event discovery in social media feeds. In *ACL*, pages 389–398. The Association for Computer Linguistics, 2011.

[7] J. Berstel. *Transductions and Context-Free Languages*. Teubner Studienbücher, Stuttgart, 1979.

[8] C. Câmpeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14(6):1007–1018, 2003.

[9] B. Carle and P. Narendran. On extended regular expressions. In *LATA 2009*, volume 5457 of *Lecture Notes in Computer Science*, pages 279–289, 2009.

[10] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, and S. Vaithyanathan. SystemT: An algebraic approach to declarative information extraction. In *ACL*, pages 128–137. The Association for Computer Linguistics, 2010.

[11] M. P. Consens and A. O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *PODS*, pages 404–416. ACM, 1990.

[12] R. Cox. Regular expression matching: the virtual machine approach. digression: Posix submatching, December 2009. http://swtch.com/ rsc/regexp/regexp2.html.

[13] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *SIGMOD Conference*, pages 323–330. ACM, 1987.

[14] H. Cunningham, D. Maynard, and V. Tablan. JAPE: a Java Annotation Patterns Engine (Second Edition). Research Memorandum CS–00–10, Department of Computer Science, University of Sheffield, November 2000.

[15] M. Dylla, I. Miliaraki, and M. Theobald. A temporal-probabilistic database model for information extraction. *PVLDB*, 6(14):1810–1821, 2013.

[16] C. C. Elgot and J. E. Mezei. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, 9:47–68, 1965.

[17] R. Fagin, B. Kimelfeld, and P. G. Kolaitis. Dichotomies in the complexity of preferred repairs. In *PODS*, pages 3–15. ACM, 2015.

[18] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Cleaning inconsistencies in information extraction via prioritized repairs. In *PODS*, pages 164–175, Snowbird, Utah, 2014. ACM.

[19] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12, 2015. Preliminary version appeared in PODS 2013.

[20] G. Fowler. An interpreation of the posix regex standard (2003), 2003. http://gsf.cococlyde.org/download/re-interpretation.tgz.

[21] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*, pages 149–158. IEEE Computer Society, 2009.

[22] R. Grishman and B. Sundheim. Message understanding conference- 6: A brief history. In *COLING*, pages 466–471, 1996.

[23] Institute of Electrical and Electronic Engineers and the Open group. The open group base specifications issue 7, 2013. IEEE Std 1003.1, 2013 Edition.

[24] V. Laurikari. Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology, 2001.

[25] X. Ling and D. S. Weld. Temporal information extraction. In *AAAI*. AAAI Press, 2010.

[26] S. Okui and T. Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. In M. Domaratzki and K. Salomaa, editors, *CIAA*, volume 6482 of *Lecture Notes in Computer Science*, pages 231–240. Springer, 2010.

[27] K. Raghunathan, H. Lee, S. Rangarajan, N. Chambers, M. Surdeanu, D. Jurafsky, and C. D. Manning. A multi-pass sieve for coreference resolution. In *EMNLP*, pages 492–501. ACL, 2010.

[28] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, pages 1033–1044. ACM, 2007.

[29] S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

[30] S. Staworko, J. Chomicki, and J. Marcinkowski. Prioritized repairing and consistent query answering in relational databases. *Ann. Math. Artif. Intell.*, 64(2-3):209–246, 2012.

[31] S. Vansummeren. Type inference for unique pattern matching. *ACM Trans. Program. Lang. Syst.*, 28(3):389–428, 2006.

[32] R. Wisnesky, M. A. Hernández, and L. Popa. Mapping polymorphism. In *ICDT*, ACM International Conference Proceeding Series, pages 196–208. ACM, 2010.

[33] H. Xu, S. P. Stenner, S. Doan, K. B. Johnson, L. R. Waitman, and J. C. Denny. Application of information technology: Medex: a medication information extraction system for clinical narratives. *JAMIA*, 17(1):19–24, 2010.

[34] D. Zelenko, C. Aone, and A. Richardella. Kernel methods for relation extraction. *Journal of Machine Learning Research*, 3:1083–1106, 2003.

[35] H. Zhu, S. Raghavan, S. Vaithyanathan, and A. Löser. Navigating the intranet with high precision. In *WWW*, pages 491–500. ACM, 2007.