Predictable and Consistent Information Extraction

Besat Kassaie

Cheriton School of Computer Science

University of Waterloo

Waterloo, Ontario, Canada

bkassaie@uwaterloo.ca

Frank Wm. Tompa

Cheriton School of Computer Science

University of Waterloo

Waterloo, Ontario, Canada

fwtompa@uwaterloo.ca

ABSTRACT

Information extraction programs (extractors) can be applied to documents to isolate structured versions of some content, that is, to create tabular records corresponding to facts found in the documents. If the data in an extracted table needs to be updated for any reason (for example, as a result of data cleaning), the source document will no longer be synchronized with the data. But *documents* are the principal medium for sharing information among humans. We therefore wish to ensure that changes to extracted tables are reflected correctly in their source documents.

In this work, we characterize extractors for which we are able to predict the effects that updates to source documents will have on extracted records. We introduce three general properties for extractors that, if satisfied, can guarantee that consistency will be maintained if the lineage of extracted records is respected when changing the documents. We propose a property verification process that uses static analysis for a substantial subset of JAPE, a well-established rule-based extraction language, and illustrate it through an example based on a freely-available extractor library.

CCS CONCEPTS

- **Information systems** → **Information extraction**; • **Applied computing** → **Document management and text processing**;
- **Security and privacy** → Data anonymization and sanitization.

ACM Reference Format:

Besat Kassaie and Frank Wm. Tompa. 2019. Predictable and Consistent Information Extraction. In *ACM Symposium on Document Engineering 2019 (DocEng '19)*, September 23–26, 2019, Berlin, Germany. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3342558.3345391>

We gratefully acknowledge financial assistance received from the University of Waterloo and NSERC, the Natural Sciences and Engineering Research Council of Canada.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).

DocEng '19, September 23–26, 2019, Berlin, Germany

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6887-2/19/09...\$15.00

<https://doi.org/10.1145/3342558.3345391>

1 INTRODUCTION

Information extraction identifies and isolates words and phrases within documents and stores them in relational tables in order to present the underlying data in structured form. Research in this area was first promoted through the Message Understanding Conferences (MUCs) [8]. The extraction process has since evolved to address specific concerns, such as efficiency and accuracy.

Our interest is in determining how changes to documents are reflected in changes to the extracted tables. More specifically, given a collection of documents and an information extraction program, we wish to determine under what conditions changing one or more occurrences of a single word or phrase p to p' in a document will cause exactly one entry in the extracted table to change from p to p' , leaving the rest of the table unchanged.

We were motivated to answer this question by the problem of applying privacy transformations to documents. Consider, for example, the problem of maintaining privacy for personal information contained in a collection of electronic health records when publishing research results derived from those records. It has been shown that simply avoiding the publication of identifiers does not protect the privacy of individuals: in the presence of other publicly available data, anonymization is vulnerable to linkage attacks [17]. The solution to this problem has been to apply *differential privacy*, which has been studied quite extensively in the context of relational tables. We are interested in the variant of differential privacy in which the data owner applies a randomized algorithm to map records in a table T to records in T' , which has the same schema as T [10]. Thereafter, T can be removed from the system, leaving researchers with full access to the synthesized table T' , a *synthetic* derivative of the input table (Figure 1). To preserve the table's utility, the synthetic values are created in such a way that a set of specified analyses on T' , such as histogram or counting queries, produce outcomes that are indistinguishable from those that would be obtained on T .

Figure 1: Creating and accessing a private table.

Our approach to protecting documents is to apply differential privacy to the table(s) obtained from a document collection through information extraction. We assume that each document D (e.g., an individual's health record) in the collection produces a single row in the extracted table T and then apply a differential privacy algorithm to form T' , thus protecting each individual document in the collection (as in Figure 1). The modified tables can be published and analyzed by untrusted parties without fearing the loss of privacy for individuals whose data is stored in the document collection.

We further assume that the researchers who are preparing tables for publication have no wish to violate individuals' privacy, but may do so unintentionally; they are assumed to be "semi-trusted." Furthermore, those researchers wish to be able to read the documents that correspond to rows in extracted tables so as to be able to interpret and validate those tables, and they are not experts on differential privacy. We therefore wish to present to those researchers a set of documents that *would have* produced the modified table had the same information extraction procedure been applied to them.

Henceforth, we are not concerned with the particulars of differential privacy, but we merely assume that entries in the extracted table are to be replaced by new values. In fact, changing a value in an extracted table might not be for privacy reasons at all, but simply as part of a data cleaning process [9] (for example, the collection of extracted records might disclose some inconsistencies among the source documents). If p is to be replaced in the table by another value p' chosen from some domain of possible values, we wish to replace one or more instances of p by p' in the corresponding location(s) of the document that produced that entry. Under what conditions does this change to the document produce exactly the expected change to the table with no other table entries affected? We say that a given extraction algorithm is *robust* if it is resilient to such modifications of text for *all* possible input document collections, *all* entries in the extracted table, and *all* values in each entry's domain.

In this work we introduce three properties required for robust extraction algorithms (Section 2). Given an extraction program, we then need to verify whether it satisfies those properties; that is, whether it is robust. In Section 4 we present a verification process that can be applied to programs expressed in JAPE [6], a well-known and commonly used rule-based language underlying the GATE information extraction system [5] and introduced in Section 3. Related work is summarized in Section 5, followed by our conclusion and outline of future work.

2 CHARACTERIZATION OF ROBUST INFORMATION EXTRACTION PROGRAMS

In this section we review the notation that we use throughout the paper. We then discuss the properties of a robust information extraction program. We show that if an information extraction program holds these properties, we can alter the document in such a way that the synthetic version of an extracted table can be extracted directly from the altered text, using the same extraction program.

2.1 Notation and Terminology

We denote the set of all possible words by W . D represents a document with length N_D , which is a sequence of words $D = \langle w_1, \dots, w_{N_D} \rangle$, $w_i \in W$. A non-empty sequence of consecutive words $\langle w_a, w_{a+1}, \dots, w_{a+m} \rangle$ (where $m \geq 0$) in a document is called a *span* and is denoted as $D[a, b]$ where $b = a + m$. Each span has a start offset a , an end offset $b = a + m$, and a length $m + 1$. ($D[a, b]$ represents an empty span when $b < a$.) $\mathcal{D}$ denotes the set of all possible documents.

A $\mathcal{T}$ -ary record consists of $\mathcal{T}$ attributes, where the i^{th} attribute A_i has domain $W_i \subseteq \bigcup_{k \in \mathbb{N}} W^k$, i.e., finite sequences of words from W . Thus the schema for a $\mathcal{T}$ -ary table T can be expressed as $T : (A_1 : W_1, \dots, A_{\mathcal{T}} : W_{\mathcal{T}})$. The universe of records $\mathcal{R}$ is the set of all possible records that could appear in table T .

In general, an information extraction algorithm is a function that maps documents to sets of records. However, in the context of differential privacy, we assume that each document belongs to one individual and is the source of exactly one record in the extracted table, i.e., we restrict ourselves to extractors¹ for which $\mathcal{X} : \mathcal{D} \rightarrow \mathcal{R}$.

Let $\mathcal{F}$ be an indexed set of *domain preserving* functions so that $\mathcal{F} = \{f_i | f_i : W_i \rightarrow W_i\}$ where $i \in [1 \dots \mathcal{T}]$ and W_i is the domain for attribute A_i . If $r = \langle v_1, \dots, v_{\mathcal{T}} \rangle$ is a record in $\mathcal{R}$, we denote the corresponding record in which the j^{th} attribute is modified as $F(r, j) = \langle v'_1, \dots, v'_j, \dots, v'_{\mathcal{T}} \rangle$, where

$$v'_k = \begin{cases} f_k(v_k) & \text{if } k = j, \\ v_k & \text{otherwise.} \end{cases}$$

We extend this notation to let $F(r) = \langle f_1(v_1), \dots, f_{\mathcal{T}}(v_{\mathcal{T}}) \rangle$.

2.2 Properties of Extractors

Strict Extractor

An information extraction program is *strict* if for every possible input document, the set of extracted values in the corresponding record is a subset of words and phrases appearing in the input. Hence, a hypothetical extractor that mines the input text and infers information that does not appear explicitly in the document is not a strict extractor. Formally, $\mathcal{X} : \mathcal{D} \rightarrow \mathcal{R}$ is *strict* if $\forall D = \langle w_1, \dots, w_{N_D} \rangle \in \mathcal{D}, \mathcal{X}(D) = \langle v_1, \dots, v_{\mathcal{T}} \rangle \implies \{v_1, \dots, v_{\mathcal{T}}\} \subseteq \{D[a, b] \mid 1 \leq a \leq b \leq N_D\}$.

If $\mathcal{X}$ is a strict extractor, we use $P_{\mathcal{X}}(D, j)$ to denote the span(s) in input document $D \in \mathcal{D}$ from which v_j is extracted. Formally, $P_{\mathcal{X}}(D, j) \subseteq \{\langle a, b \rangle \mid D[a, b] = v_j\}$, i.e., a subset of starting and ending positions in D where the sequence of words in the span matches v_j . We assume that the spans in $P_{\mathcal{X}}(D, j)$ do not overlap.²

Computable Extractor

A strict extractor is *computable* if for all possible input documents and corresponding extracted attributes, we have access to positions from which the attributes are extracted. Formally, $\forall j \in [1 \dots \mathcal{T}]$

¹We use "information extraction program" and "extractor" interchangeably.

²Only for very restricted values of v_j could the spans, in fact, overlap.

and $\forall D \in \mathcal{D}$, $\mathcal{X}$ is *computable* if $P_{\mathcal{X}}(D, j)$ is explicit (given) or provided as a side-effect of running the extractor on D .

Stable Extractor

Let $\mathcal{X}$ be a strict and computable extractor, $D = \langle w_1, \dots, w_{N_D} \rangle$ be a document, A_j be an extracted attribute with value v_j , and $P_{\mathcal{X}}(D, j) = \{\langle a_i, b_i \rangle\}$ be the set of integer pairs denoting spans in D from which v_j is extracted, where $1 \leq a_1 \leq b_1 < a_2 \leq b_2 < \dots < a_k \leq b_k \leq N_D$. We define $g(D, j)$ to be the result of substituting $f_j(v_j)$ for v_j in all spans identified by $P_{\mathcal{X}}(D, j)$. Formally,

$$g(D, j) = D[1, a_1 - 1] \bullet f_j(v_j) \bullet D[b_1 + 1, a_2 - 1] \bullet f_j(v_j) \bullet \\ D[b_2 + 1, a_3 - 1] \bullet f_j(v_j) \bullet \dots \bullet D[b_k + 1, N_D]$$

where $\bullet$ indicates concatenation.

An information extraction algorithm is *stable* if $\forall D \in \mathcal{D}$ and $\forall j \in [1 \dots \mathcal{T}]$ we have $\mathcal{X}(D) = r \implies \mathcal{X}(g(D, j)) = F(j, r)$. Thus, with a stable extractor, changing values in appropriate positions in a document affects only the expected attribute in the extracted record.

Theorem. Consider any strict, computable, and stable extractor $\mathcal{X} : \mathcal{D} \rightarrow \mathcal{R}$, any indexed set of domain preserving functions $\mathcal{F} = \{f_i \mid f_i : W_i \rightarrow W_i, \text{ where } i \in [1 \dots \mathcal{T}]\}$, and any document $D \in \mathcal{D}$. For all $j \in [1 \dots \mathcal{T}]$, substituting $f_j(v_j)$ for v_j in all spans identified by $P_{\mathcal{X}}(D, j)$ produces $D_{\mathcal{F}}^j$ in such way that $F(\mathcal{X}(D)) = \mathcal{X}(D_{\mathcal{F}}^j)$.

PROOF. $\mathcal{X}$ is strict, therefore all v_j in r occur in D . Being a computable extractor, $P_{\mathcal{X}}(D, j)$ is known for every v_j in r . So the locations of all spans in D that need to be modified is accessible to the procedure. Finally, $\mathcal{X}$ is stable, so any substitutions corresponding to each attribute v_j affect only the j^{th} attribute in r . $\square$

3 OVERVIEW OF JAPE

The interpretability of rules makes rule-based information extraction systems suitable for domains in which a user's confidence in the validity of the extraction algorithm is critical [4]. GATE [5] is a commonly used rule-based information extraction system, which identifies critical pieces of a document using a grammar written in JAPE [6]. To simplify our discussion, we assume a simplified environment for running JAPE: a simple tokenizer is executed before JAPE begins and a simplified output generator is executed after JAPE completes its computations. (In practice, when using GATE, tokenizers and output generators can be far more complex.) Thus our extractors include the following stages:

- (1) Given an input document, a tokenizer first constructs a linear annotation graph in which the edges correspond to sequential spans of text (e.g., words). A Jape program then describes how to traverse and modify the annotation graph.
- (2) Running a JAPE program involves executing a set of matching *rules*, written as regular expressions over annotations that label edges in a rooted directed acyclic graph. Rules are organized into a sequence of *phases* that run consecutively in the order that they appear in the program. In each phase, JAPE interprets the rules in that phase to traverse the annotation graph (from its unique source), identifying possible matches. This is explained

in further detail below. Each phase might modify the annotation graph by adding and/or removing annotations.

- (3) After all phases have been executed, some of the annotation labels (herein called *extracted annotations*) are interpreted as designations for spans of text to be extracted. For simplicity, we assume that a JAPE extractor produces exactly one record per document, that is, $\mathcal{X} : \mathcal{D} \rightarrow \mathcal{R}$. Running the extractor over a document D produces a record in which each attribute is populated with the value from D matched by the corresponding extracted annotation (or null if no span is bound to that annotation). (If an extracted annotation is bound to more than one span of text in a document D , we assume that all matched spans for that annotation contain the same word or phrase.)

Each JAPE rule is composed of a *pattern* and an *action*, as depicted in Figure 2. The pattern appears to the left of the separator $\rightarrow$ and describes a regular expression over annotations. These regular expressions are converted to finite state transducers, which consume the input and assign spans of text to *binding variables* once a match happens. The part of the left-hand side of a rule that is bound to a variable is called a *rule nest*.

For example, in Figure 2, S_i represents the i^{th} group of constraints, S_2 is a rule nest bound to the variable α , and S_4 is the rule nest bound to β . A JAPE rule annotates spans of text not only based on constraints in the nests, i.e., S_2 and S_4 but also based on constraints over the contexts, i.e., S_1, S_3 , and S_5 .

$$S_1 (S_2) : \alpha \ S_3 (S_4) : \beta \ S_5 \rightarrow : \alpha . A, : \beta . B$$

Figure 2: A prototypical JAPE rule with two nests.

In the action part of a *simple rule* (appearing after $\rightarrow$), the user can include instructions to associate annotations with the spans of text that are marked by binding variables and add them to the annotation graph. (It is possible to write custom Java code to perform more complex operations over annotations, but in this work we permit simple rules only.) In Figure 2, the rule adds both “A” and “B” to the annotation graph only if all the S_i match the input.

In addition to rules, two declarations for a phase are essential to our work: *Input* and *Policy*. Input enumerates the annotation types that are available to the rules in a phase: only those edges in the annotation graph that are labelled by input types are visible to the rules in each phase. Thus, for example, two annotations are deemed to be adjacent if they cover two spans of text with no annotation in the *Input list* covering a span that starts between them. For the rule in Figure 2 to match, therefore, the S_i must refer to annotations listed in the Input list only and no other annotations included in that list must intervene.

Policy specifies the order in which spans of text are considered and where scanning resumes after a rule is fired. Thus the policy determines the strategy to be taken to pick a match when more than one span can be matched and when matches might overlap. In particular, for the rule in Figure 2 to match, the policy must not dictate that some other rule in the phase takes precedence. JAPE offers five policies: *Once*, *First*, *Appelt*, *Brill*, and *All*.

```

Phase:   UrlPre
Input:   Token SpaceToken
Options: control = appelt
Rule: Urlpre
( ( ({Token.string=="http"} | {Token.string=="ftp"})
  {Token.string==":"}{Token.string=="\/"}{Token.string=="\/"}
) | ({Token.string=="www"}{Token.string=="."}) ):urlpre
--> :urlpre.UrlPre = {rule = "UrlPre"}

```

Figure 3a: UrlPre phase defined in ANNIE's url_pre.jape³

With the Once and First policies, spans are tested from shortest to longest; whenever a match for a rule is found, the rule matching that span is fired and the corresponding span is marked with the relevant annotation. Thus longer matches are ignored. In the Once policy, the phase exits after firing of a rule (so scanning for the *next* phase resumes from the root of the annotation graph), whereas the First policy continues to search for additional matches in the *current* phase, starting from the end of the matched span. For both policies, if more than one rule matches the same shortest span, the JAPE processor arbitrarily chooses one of them to be fired and the others are ignored. (Multiple runs over the same input could therefore produce non-identical extractions.)

With the Appelt policy, spans are tested from longest to shortest, and the longest matched span causes a rule to be fired. (Thus shorter matches are ignored.) When multiple rules in a phase can be fired for the same longest span of text, JAPE chooses a rule based on priority. A rule's priority with respect to other rules is determined first from the declared priority number in the rule (higher numbers having higher priority). If several matching rules have the same priority number, the rule that appears earlier in the phase fires. After firing a rule, scanning for additional matches in the current phase starts from the end of the matched span.

The Brill policy allows *all* the rules that match a span of any length starting from some point to be fired. As a result, overlapping annotations can be created within a single phase. After firing all matching rules, scanning for additional matches in the current phase starts from the end of the longest matched span.

Finally, the All policy allows all possible rules to fire, regardless of the starting point and the span of text that matches.

A simple JAPE program with four phases is shown in Figure 3. The first two phases (Figures 3a³ and b⁴) are copied from ANNIE, a ready-to-use IE bundle that is implemented and distributed with GATE [5]. The first phase has one rule that matches “*Tokens*” and “*SpaceTokens*” (assumed here to be annotations that are created before JAPE begins) and uses the Appelt policy (longest match). It recognizes the strings “http://”, “ftp://”, and “www.” as common

```

Phase:   Url
Input:   Lookup SpaceToken Token UrlPre
Options: control = appelt
Rule: Url1
Priority: 50
({UrlPre}({Token})[1,7]):urlAddress ({SpaceToken})
--> :urlAddress.Url = {kind="urlAddress", rule="Url1"}
Rule: Url2
Priority: 100
({UrlPre} ({Token})[1,7]{Token.string=="."}
 {Lookup.majorType==country_code}
--> :urlAddress.Url = {kind="urlAddress", rule="Url2"}
Rule: UrlContext
Priority: 20
({Token.string=="at"}{Token.string==":"})
( ({Token.orth==lowercase}|{Token.orth==upperInitial} |
  {Token.kind==number} |{Token.kind==punctuation} |
  {Token.kind==symbol} |{Token.string=="."})+
  {Token.string=="."}
  ({Token.orth==lowercase}|{Token.orth==upperInitial} |
  {Token.kind==number} |{Token.kind==punctuation} |
  {Token.kind==symbol} |{Token.string=="\/"}
  {Token.string=="."})+ ):urlAddress
--> :urlAddress.Url={kind="urlAddress", rule="UrlContext"}
Rule: UrlGuess
Priority: 10
// token(s) + url_ending e.g. gate.ac.uk
( ({Token}{Token.string=="."})+
  {Lookup.majorType==url_key}{Token.string=="."}
  {Lookup.majorType==country_code}
):urlAddress
--> :urlAddress.Url={kind="urlAddress", rule="UrlGuess"}

```

Figure 3b: Url phase defined in ANNIE's url.jape⁴

```

Phase:   Names
Input:   Token
Options: control = appelt
Rule: capitalized
( ({Token.orth==upperInitial})+ ):mark
--> :mark.Capitalized={rule="capitalized"}

```

Figure 3c: Spans of capitalized words

```

Phase:   NameAndURL
Input:   Capitalized Url
Options: control = first
Rule: final
(Capitalized}):name ({Url}):page
--> :name.Name={rule="final"}, :page.Page={rule="final"}

```

Figure 3d: Collocations of names and URLs

indicators of the start of a URL, and annotates these as “*UrlPre*”. The second phase has four rules that match the annotations created during the first phase, as well as “*Lookup*” (a match in an associated dictionary), “*Token*”, and “*SpaceToken*”, and again uses the Appelt policy. URLs that start with a string recognized in the first phase, followed by one to seven tokens with no intervening spaces, and possibly ending with either a “*SpaceToken*” or a period and a country code found in the dictionary are recognized by the first two rules. The third looks for the string “at:” and matches whatever follows

³Copyright (c) 1998-2004, The University of Sheffield. This file is part of GATE (see <http://gate.ac.uk>), and is software, licenced under the GNU Library General Public License, Version 2, June 1991 (in the distribution as file <http://gate.ac.uk>licence.html), and also available at <http://gate.ac.uk>gatelicence.html). Diana Maynard, 19 April 2001 \$Id: url_pre.jape 5921 2004-07-21 17:00:37Z akshay

⁴Copyright (c) 1998-2004, The University of Sheffield. This file is part of GATE (see <http://gate.ac.uk>), and is software, licenced under the GNU Library General Public License, Version 2, June 1991 (in the distribution as file <http://gate.ac.uk>licence.html), and also available at <http://gate.ac.uk>gatelicence.html). Diana Maynard, 02 Aug 2001 \$Id: url.jape 18155 2014-07-04 13:22:03Z dgmaynard

Figure 4: Verifying robustness for a JAPE program.

as long as it includes symbols that might appear in a URL. The fourth rule matches a string that is a sequence of token-period pairs, followed by a symbol marked in the dictionary as a “url_key” (e.g., ac), followed by a period and a country code. The last two phases (Figure 3c and d) have been added for this example. The third phase uses the Appelt policy to find maximal sequences of alphabetic tokens that begin with an uppercase letter and otherwise contains only lowercase letters. The final phase uses the First policy (shortest match) to find strings that look like names followed by a URL (ignoring all annotations that are neither “Capitalized” nor “Url”). We assume that “Name” and “Page” are extracted annotations, and the spans marked by their binding variables are the portions of text that are subject to update.

4 VERIFICATION OF JAPE PROGRAMS

Given a JAPE program and a collection of domain preserving functions, the verification process depicted in Figure 4 determines whether the JAPE program satisfies the proposed properties, i.e., whether the extractor is robust.⁵ JAPE phases run sequentially, and in each phase three factors specify whether a rule fires: the phase’s policy, the rule’s constraints and priority, and the input text. However, so that robustness remains a property of a program and its accompanying domain preserving functions, the verification process needs to be agnostic to the specifics of the input text.

Ideally, verification would identify every robust JAPE program as robust and every non-robust JAPE program as non-robust, for which it would need to investigate necessary and sufficient conditions. As a start, the focus of this paper is to demonstrate that our verification process maintains sufficient conditions for a robust extractor. Hence, we show that our verification process declares a program and accompanying domain preserving functions to be robust only if it allows updates for all possible input texts.

In the remainder of this section, we overview the techniques by which our verification process determines the robustness of a JAPE program. We explain our method over restricted JAPE rules that are a simple variant of Figure 2 with one nest and left and right contexts. We show cases that comply with or violate a property using examples. Of course, a JAPE program can be more complex than our examples, but the discussed cases cover the fundamentals of our verification process.

4.1 Strict JAPE Program

Every JAPE program that contains only simple rules and populates tables using extracted annotations has the strict property.

⁵For JAPE, being computable need not be verified: see Section 4.2.

If a simple JAPE rule fires, assigning annotations to spans of the input document is the only action that happens. As a result, only a subset of words that appears in the document will be extracted. Hence our verifier merely needs to examine each rule to determine whether it is simple. (JAPE programs that do not comprise only simple rules might also be strict, but that is not our concern in showing sufficiency.) The JAPE program in Figure 3 is strict.

4.2 Computable JAPE Program

Each binding variable is *unique* within the scope of a rule’s pattern. If a rule fires, its binding variables provide the start and end offsets of the corresponding spans to the action, where they can be associated with their annotations. Thus for every value in an extracted record, v_j , these offsets can be traced back to determine $P_X(D, j)$ and can be used to locate the extracted words and to generate $g(D, j)$. As a result, every strict JAPE program is computable.

4.3 Stable JAPE Program

The stability property reflects how an extracted record is affected when substituting attribute values in the source document. Therefore we study whether running a strict (and computable) JAPE program over $g(D, j)$ for all $j \in [1, \dots, \mathcal{T}]$, extracts a *correctly modified* record. Unwanted side-effects include extracting a value other than v'_j as A_j and extracting a value other than v_k as A_k for some $k \neq j$. We discuss under what conditions undesired effects might occur for a computable JAPE program.

Domain Inconsistency

In JAPE, the constraints on annotations in each rule’s pattern define a domain. We require that, for each domain-preserving function in $\mathcal{F}$, the domain of f_j is a subset of the domain formed by the rule constraints associated with the binding variable corresponding to the annotation extracted for attribute A_j . For example, in Figure 3d the domain of “Name” is all sequences of one or more tokens, each with an initial uppercase letter followed by all lowercase letters. This domain originates from constraints that appear in the third phase.

If the extracted “Name” (*John Diefenbaker*) is replaced by a member of the domain that meets the constraints (e.g., *Stawamus Chief Mountain*) and the extracted “Page” (*www.the.chief.on/home*) is replaced by a member that meets its constraints, the altered values get extracted as expected (Figure 5b). On the other hand, if the name were replaced by *Toronto’s Chief of Police*, the replacement does not get extracted properly, as the replacement text is outside the domain formed by the rule constraints (Figure 5c). Domains imposed by JAPE rules are derived from one or more rules appearing in one or more phases, but they can be captured algorithmically. As a result, we can ensure that each domain preserving function in $f_j \in \mathcal{F}$ avoids domain inconsistency.

Problematic Overlaps

Domain consistency alone does not guarantee extracting a correct record from $g(D, j)$. Sometimes a rule *overlaps* itself or other rules. *Overlap diagnosis* is used to investigate stability. For illustrative

(a)	...	Note	that	John	Diefenbaker	owns	page	http://www.the.chief.on/home	...
				[Name]				[Page]	
(b)	...	Note	that	Stawamus	Chief	Mountain	owns	page	http://www.the.chief.bc/home
					[Name]				[Page]
(c)	...	Note	that	Toronto's	Chief	of	Police	owns	page
							[Name]		[Page]
(d)	...	Note	that	John	Diefenbaker	owns	page	http://www.john.on/home	...
				[Name]				[Page]	
(e)	...	Note	that	John	Diefenbaker	owns	page	http://www.the.chief.ca/home	...
				[Name]				[Page]	

Figure 5: Extractions from input texts using JAPE program in Figure 3.

Figure 6: Example schematic for overlap analysis.

purposes, we assume that each rule has one nest S_{i2} , a left context matching S_{i1} , and a right context matching S_{i3} (Figure 6a). The corresponding spans in the text are represented by n_i , lc_i , and rc_i , respectively. For simplicity we assume that the right-hand side of the rule annotates an extracted annotation specified by the only nest in the left-hand side. Therefore n_i is the span from which the value for A_i is extracted.

When updating a document, we replace an extracted value by a new value in the document locations where the prior value *and*, we expect, the new value after substitution are obtained. Such a replacement will violate stability when (a part of) a rule's nest can serve as a different part of the same rule's pattern or as a part of some other rule's pattern. Therefore the nest for an extracted annotation needs to be involved in a problematic overlap in the patterns, but the overlap with the conflicting rule can occur anywhere in the rule (see, for example, Figure 6b).

Assume that the input program contains only one phase consisting of two (not necessarily distinct) rules of the form presented in Figure 6a. Figure 6b illustrates a situation in which the concatenation of lc_2 , n_2 , and a prefix of rc_2 overlaps the concatenation of the left and right context and the nest in R . If R matches a hypothetical input on which R' does not match, JAPE will extract the span associated with n_1 as value v_1 in A_1 . Assume the document is altered so that the text corresponding to that span is replaced by a value v'_1 , and that value causes the formation of $lc_2-n_2-rc_2$. If the rule is in a phase governed by the Appelt policy, the span earlier marked by n_1 , i.e. v'_1 , will no longer be extracted; there is a longer match available through R' . Such overlapping rules disqualify the extractor from being stable.

As a concrete example of a problematic overlap, consider again Figure 5a in which *www.the.chief.on/home* is extracted as “Page”. Replacing this text by *www.john.on/home* does not result in extracting a corresponding record even though it does not violate domain consistency (Figure 5d). Because the replacement text includes seven tokens, it is now available to match the prefix starting with *http://* and continuing to the space token after the address (using *Url1*). As a result, the extracted text now includes the string *http://*, whereas it did not before the update. Similarly, replacing *www.the.chief.on/home* by *www.the.chief.ca/home* now allows *Url2* to match starting earlier in the text, again causing an unexpected value to be extracted (Figure 5e).

To summarize, the verification process examines a JAPE program to find at least one case of domain inconsistency or problematic overlap. In the absence of such a case, the input program is stable. We next present how we explore the space of potential overlaps.

Rubber Band Analysis

For every JAPE program an overlap diagnosis tool needs to investigate pairs of extracted annotations by examining all possible realizations of problematic cases in the space of potential overlaps.

We fix a region of hypothetical text, depicted as a block in Figure 7 over which a hypothetical rule, R , has already fired. We assume that the span identified as n is extracted as attribute A and that the text within that span is to be replaced by some other value from the same domain. Our aim is to examine all potentially problematic instances of a second (not necessarily distinct) rule, R' , in which the modified text causes R' to fire instead of R firing. For this to happen, at the very least, the span matched by R' needs to overlap the span matched by R , and more specifically, n .

Our approach exploits the boundaries of the spans matched by both R and the potentially competing rule R' . The boundaries of the left and right contexts and the nests in both rules are labelled as in Figure 7. The spans matched by R are assumed to be fixed, with boundaries A, B, C, and D, which determine five regions, as follows:

- A region to the left of boundary A
- AB region between boundaries A and B
- BC region between boundaries B and C
- CD region between boundaries C and D
- D+ region to the right of boundary D

Figure 7: Visualizing the rubber band analysis.

Together with the four boundaries (A, B, C, and D), this gives nine locations to consider for each of the four boundaries of the span matched by R' . To investigate the space of all possible overlaps, the boundaries for the spans matched by R' can be thought to move from left to right over the text (as if we fixed the leftmost edge of a rubber band with markings for A' , B' , C' , and D' to the start of the input text and then stretched the rubber band to the right). Thus, with the left-to-right order of all boundaries remaining unchanged, we can consider each possible location for each boundary in turn. Figure 7 depicts a situation in which A' and B' are both in region $-A$, C' is in region AB , and D' is in region CD ; When considering all possible overlaps, we can immediately eliminate ones that do not involve the update target, i.e., the region between B and C (the span annotated by the nest in R). Thus, A' must be to the left of C (i.e., in regions $-A$, AB , or BC or at boundary points A or B), and D' must be to the right of B (i.e., in regions BC , CD , or $D+$ or at boundary points C or D).

Finally, we can also eliminate cases that are not problematic for the given phase's policy. For example if A' coincides with A, D' is in the region $D+$, text were substituted for the span marked as n , and this allowed the match of R' , it would be problematic under the Appelt policy, but not under the First policy, since the span matched by R' would be longer than the one matched by R .

The rubber band analysis must be applied to each pair of rules that involve extracted attributes. However, mapping those rules to spans of text requires that the effects of prior rules, those creating the input annotations used by those rules, be taken into account.

As each potentially problematic overlap is identified, our diagnosis tool uses finite state automata to investigate whether the domains for matching rule R include values that can also be interpreted to match R' with the given overlap.

Transformation to non-nested rules

The rubber band analysis accepts a pair of simple, unnested rules. We now present an algorithm to transform any JAPE phase containing nested rules to a sequence of "flat" phases, in which no rules are nested. To explain the process of transforming nested rules to non-nested ones, we first introduce a few definitions.

Definition 4.1. If v is a binding variable, then $nest(v)$ is the corresponding rule nest and $annot(v)$ is the corresponding annotation.

Definition 4.2. The *depth* of a rule nest is the number of binding variables within which the rule nest appears. (Thus, the depth of a

nest that appears within a binding variable that is not nested within another rule nest is 0.)

Definition 4.3. The maximum depth of all the nests in a rule is the *rule depth*.

Definition 4.4. The greatest rule depth of all the rules in a phase is the *phase depth*.

Definition 4.5. A phase is *flattened* if its phase depth is 0.

For example, for the nested rule depicted in Figure 8, nested binding variables include θ , γ , and η . The rule has depth three, and the nest bound to γ has depth two.

The transformation process in Algorithm 1 will be applied over all phases in Φ_0 , and a new sequence of phases Φ will be produced. For simplicity, we will treat a phase and the sequence of rules contained in the phase interchangeably wherever there is no confusion.

If a JAPE phase has at least one rule with depth greater than 0, it will be replaced by a sequence of new phases; otherwise it stays intact. The output of Algorithm 2 for a single phase with depth d as the input is a sequence of phases $\langle \mathcal{P}'_0, \mathcal{P}'_1, \mathcal{P}'_2, \dots, \mathcal{P}'_{2 \times d} \rangle$, which replaces the input phase. $\mathcal{P}'_0$ will be assigned the same policy as that of the input phase $\mathcal{P}$, and the *All* policy will be assigned to all other generated phases.

Algorithm 2 performs three major operations over the sequence of rules in the input phase:

- Lines 12-13 create new rules that are responsible for marking parts of the text that satisfy the constraints found in nests at depth $d - 1$. A new rule, ρ , is created for each such nest, B_j . For every binding variable in B_j , a temporary annotation, T_{V_n} , is introduced to mark part of the text that might eventually contribute to the annotation for B_j . All these new rules are added to a new phase of *BaseRules*.
- Lines 14-16 recover the original annotations from the generated temporary annotations. Only those temporary annotations that appear inside $annot(B_j)$ should be recovered; others are spurious. The resulting rules are added to a phase called *RecovRules*.
- The depth reduction process removes all the outermost binding variables for nests of depth d as well as their corresponding annotations. If the depth of a rule is less than the phase depth, d , it is not modified and will be added to *RdcRules* (Line 18). Input rules with depth d become rules with depth $d - 1$ and

$$S_1 (S_2):\alpha S_3 (S_4 (S_5 (S_6 (S_7):\eta S_8):\gamma S_9):\theta):\beta \rightarrow \text{annotate spans corresponding to } \alpha, \beta, \theta, \gamma, \text{ and } \eta.$$

Figure 8: A prototypical nested JAPE rule.

Input: Φ_0 , a sequence of $\mathcal{K}$ phases $\langle \mathcal{P}_1, \dots, \mathcal{P}_{\mathcal{K}} \rangle$
Output: Φ , a sequence of flattened phases $\langle \mathcal{P}'_1, \dots, \mathcal{P}'_{\mathcal{K}'} \rangle$, $\mathcal{K}' \geq \mathcal{K}$
 $\Phi = \langle \rangle$ // an empty sequence
for $k = 1$ **to** $\mathcal{K}$ **do**
 $\Phi.append(\text{Expand}(\mathcal{P}_k))$ // uses Algorithm 2 to flatten each phase
end
return Φ

Algorithm 1: Flattening a sequence of phases.

Input: $\mathcal{P}$, a sequence of rules $\langle R_1, \dots, R_m \rangle$
Output: A sequence of phases $\langle \mathcal{P}'_0, \mathcal{P}'_1, \mathcal{P}'_2, \dots, \mathcal{P}'_{2 \times \mathcal{P}.depth} \rangle$

```

1   $d = \mathcal{P}.depth;$ 
2  if  $d == 0$  then
3      return  $\langle \mathcal{P} \rangle$ 
    /* a sequence having one phase already flattened */
4  else
5       $RdcRules = \langle \rangle;$ 
6      for  $i = 1$  to  $m$  do
7          if  $R_i.depth == d$  then
8               $\Psi = \text{binding variables at depth } d;$ 
9               $\mathcal{B} = \text{binding variables at depth } d - 1;$ 
10              $BaseRules = \langle \rangle; RecovRules = \langle \rangle;$ 
11             foreach  $B_j \in \mathcal{B}$  do
12                 Let  $\mathcal{V} = \{V_1, \dots, V_k\}$  be the binding variables
13                 appearing in  $nest(B_j)$ ;
14                  $\rho = nest(B_j) \rightarrow V_1.T_{V_1}, \dots, V_k.T_{V_k};$ 
15                 /*  $T_V$  are new temporary annotations */
16                  $BaseRules.addRule(\rho);$ 
17                 foreach  $V_n \in \mathcal{V}$  do
18                      $\rho = (T_{V_n} \text{ within } annot(B_j)):x$ 
19                      $\rightarrow x.annot(V_n);$ 
20                      $RecovRules.addRule(\rho)$ 
21                 end
22             end
23              $R_i = \text{Omit } \Psi \text{ and corresponding annotations from } R_i$ 
24             /* a rule with depth d-1 */
25         end
26          $RdcRules.addRule(R_i)$ 
27     end
28     return  $\text{Expand}(RdcRules)$ 
29      $.append(\text{newPhase}(BaseRules))$ 
30      $.append(\text{newPhase}(RecovRules))$ 
31 end
```

Algorithm 2: Expand($\mathcal{P}$).

are also added to *RdcRules* (Line 17 and 18). *RdcRules* is then processed recursively before the final sequence of phases is returned (Line 19).

Thus, with the aid of Algorithm 2, Algorithm 1 produces a sequence of JAPE phases, all having depth zero, that is semantically equivalent to the original sequence of phases with arbitrary depth. Figure 9 illustrates a simple example in which the input is a single phase having depth 1 and the output is a sequence of three phases.

After flattening, the resulting sequence can be subjected to rubber band analysis, as previously described.

4.4 Applicability of JAPE Verification

ANNIE, a commonly used basis for information extraction using GATE, includes a tokeniser, dictionaries, and a set of JAPE rules [5]. ANNIE's rules are organized in 21 phases to identify entities such as "Person" and "Address". Our sufficiency analysis can be used to verify the 11 phases that contain simple rules only. For illustration, we report the outcome of applying the proposed verification process over the two phases for which the goal is to identify URL instances in a document (Figures 3a and b).

The extractor in Figure 3 relies on the flow of annotations (and associated phase dependencies) shown in Figure 10. When testing for overlaps, we need to apply the input list, rule structures, and various prioritization policies during the push down process from phases *UrlPre* to *NameAndURL*.

The given phases do not include any nested rule, so we can apply the overlap diagnosis tool without expanding the phases.

The analysis reveals that *Url* contains rules with problematic overlaps, and therefore it is not verified as stable. For instance, *url1* (and similarly *url2*) is written in such a way that a string can be interpreted as a "*UrlPre*" even if it is preceded by another "*UrlPre*". Furthermore, imposing a limit on the number of tokens, i.e. [1,7], prevents the Appelt policy from accepting what is truly the longest match. Therefore, there exists a non-empty set of inputs on which *url1* (*url2*) matches starting from two different offsets, one violating the length limit. As illustrated in Figure 3, the match that starts earlier and violates the length constraint possibly complies with the length limit after substitution. Thus the earlier span is accepted, and an unexpected value gets extracted.

The proposed verification process does not offer possible remedies to make a program robust, although this is a worthwhile research question. After investigating the program manually, it turns out that applying a few small changes can make the program robust while (likely) complying with the intended extraction semantics:

- (1) We prepend a new phase called *UrlIntro* that matches the string at: (and could be extended to match other introductory strings as well, if desired).

Figure 9: An example of flattening.

Figure 10: Annotation Flow Graph.

- (2) To prevent overlap between *Url1* and *UrlGuess*, we modify $\{Token\} \{Token.string == "."\}$ in *Url1* to $\{Token, !UrlPre\} \{Token.string == "."\}$.
- (3) We augment the rule in *UrlPre* to allow two additional prefixes: "http://www." and "ftp://www." explicitly.
- (4) In *Url*, we add "*UrlIntro*" as an input annotation, remove the length limit in the first rule, and remove the second rule completely. We also change each occurrence of an unrestricted $\{Token\}$ in any rule to disallow matches to "*UrlIntro*" or "*UrlPre*". As a result, for example, the first rule becomes:

```

({UrlPre}({Token, !UrlIntro, !UrlPre})+{Token.string == "."}
{Token, !UrlIntro, !UrlPre})+:urlAddress {SpaceToken}
-->:urlAddress.Url={kind="urlAddress", rule="Url1"}

```

With these few changes, the program to find collocations of names and URLs will pass the overlap analysis; as a result updates to an extracted table can be reliably propagated back to the source documents.

5 RELATED WORK

Information extraction. Expectations from extractors have risen as requirements have become more diversified, from the point that there were no criteria to evaluate their performance [7] to the point that extraction algorithms need to work under various stresses such as noisy data, low response time, and diverse types of input and output [16]. The problem closest to our situation is continuous adaptation of extractors as their information sources change. Lerman et al. [12] monitor updates on information sources for a specific class of extraction algorithms (wrappers) and rebuild the extractor if the performance decreases due to the updates over their sources. For our application of interest, however, we identify the extractors that can tolerate supervised updates over the sources without applying any adjustments to the extractor itself. Again, our goal is to modify the document, not the extractor.

Fine-grained data lineage. Data lineage, or provenance, has been defined and formalized for structured and semi-structured data [1, 3]. Given a value, which is the outcome of executing a well-defined query over some data sources, often relational tables, provenance determines three aspects related to the value: data points in the source that contribute to form the value, the way that data points collaborate to produce the value, and the exact location(s) in the data source from which the value originates. The last aspect is similar to the notion of lineage that we use in this work, i.e., we require the extractor to provide the positions in a document from which a value is extracted.

Provenance-based techniques have also been applied to information extraction problems. Roy et al. [15] propose a provenance-based technique to improve the quality of extraction by refining the dictionaries that are used in a rule-based extraction system. A set of

entries from the dictionaries that have been involved in generating the output are analyzed to determine which should be removed to improve the extractor's performance most. In other work, Liu et al. [13] use provenance techniques to determine the most effective rule refinements, i.e., those that result in removing undesirable tuples and keeping correct ones. Chai et al. [2] examine the provenance of a multi-stage extraction program that can include relational operators on intermediate tables. Users' feedback is expressed as updates over tuples that appear at any stage of the extraction process, and these updates are translated into modifications of the corresponding extraction program.

Static analysis of programs using regular languages. Our verifier uses finite-state automata to determine whether an extractor is stable. Similar static analyses of regular expressions have been used in diverse areas, including access control and feature interactions. For example, Murata et al. [14] propose an automaton-based access control mechanism for XML database systems. Regular expressions are derived from given queries, access-control policies, and schemas. Based on characteristics of the derived automata, element/attribute level access requests by queries are determined to be either granted, denied, or statically indeterminate, independently of any actual input XML documents. An event-based framework is introduced by Kin et al. [11] for developing and maintaining new gestures that can be used in multitouch environments. Using that framework, application developers express each gesture as a regular expression over some predefined touch events. Regular expressions associated with gestures are then statically analyzed to identify possible conflicts between various gestures.

6 CONCLUSIONS

This paper characterizes extraction algorithms that are resilient to changes in their source documents intended to reflect predetermined changes to the extracted table. We first introduced and formalized the notion of robust information extraction algorithms and proposed three properties for such extractors. We further proposed a straightforward algorithm that modifies the input document considering the robust extractor and a set of domain preserving functions. The modified document can be fed into the extractor to produce the expected synthetic table. We outlined the essentials for designing a verification process for robustness of programs written in a significant subset of the commonly used JAPE language.

We have made some simplifying assumptions in our work, each of which can be modified or eliminated to expand the class of programs deemed to be robust. For example, we have assumed independence between extracted attributes, thus requiring that at most one extracted value can be affected by each change in the source document. What if instead we are given constraints among the attributes, such as A_2 and A_5 must be identical or must have (computably) dependent values? We have also assumed that each table attribute can be given a value from a single span in the input document. What if several words or phrases can be combined to create an extracted value? Loosening our simplifying assumptions might result in being able to verify more useful extractors.

We have presented a property verification process applicable to a subset of all possible JAPE programs. Currently we are formalizing the complete verification process and proving its correctness. However, we will not address other JAPE capabilities, such as allowing annotations to be removed from the graph (which may not be problematic) or using Java code to describe a rule's actions (which, in general, will make stability undecidable).

We have developed a set of sufficient properties for robust JAPE programs, but we have not investigated which properties might be necessary. Thus we can state with assurance that some JAPE program are robust, but we cannot verify that a JAPE program is *not* robust. We wish to explore necessary properties, approaches to verifying those properties for JAPE, and whether some programs that cannot be verified as robust can be transformed into ones that possess the required properties of robustness.

Finally, we need to develop verification tools for other forms of extractors, including those crafted in other extraction languages and those based on machine learning technology.

REFERENCES

- [1] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Proc. 4th Int. Conf. on Database Theory (ICDT 2001)*. 316–330. https://doi.org/10.1007/3-540-44503-X_20
- [2] Xiaoyong Chai, Ba-Quy Vuong, AnHai Doan, and Jeffrey F. Naughton. 2009. Efficiently incorporating user feedback into information extraction and integration programs. In *Proc. 2009 SIGMOD Conf. Mgmt. of Data*. ACM, 87–100.
- [3] James Cheney, Laura Chiticariu, and Wang Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2009), 379–474.
- [4] Laura Chiticariu, Yunyao Li, and Frederick R. Reiss. 2013. Rule-Based Information Extraction is Dead! Long Live Rule-Based Information Extraction Systems!. In *2013 Conf. on Empirical Methods in Nat. Lang. Proc., (EMNLP 2013)*. 827–832.
- [5] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. 2002. A framework and graphical development environment for robust NLP tools and applications. In *40th Ann. Meet. Assoc. Comp. Ling.* 168–175.
- [6] Hamish Cunningham, Diana Maynard, and Valentin Tablan. 2000. *JAPE: a Java annotation patterns engine (Second Edition)*. Technical Report CS-00-10. Dept. Comp. Sci., Univ. Sheffield.
- [7] Robert Gaizauskas and Yorick Wilks. 1998. Information extraction: Beyond document retrieval. *Journal of Documentation* 54, 1 (1998), 70–105.
- [8] Ralph Grishman and Beth Sundheim. 1996. Message Understanding Conference-6: A Brief History. In *Proc. 16th Int. Conf. Comp. Ling. (COLING 1996)*. 466–471.
- [9] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning*. Morgan and Claypool.
- [10] Shiva Prasad Kasiviswanathan, Homin K. Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam D. Smith. 2011. What Can We Learn Privately? *SIAM J. Comput.* 40, 3 (2011), 793–826.
- [11] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. 2012. Proton: multitouch gestures as regular expressions. In *ACM Conf. on Human Factors in Computing Systems (CHI '12)*. 2885–2894.
- [12] Kristina Lerman, Steven Minton, and Craig A. Knoblock. 2003. Wrapper Maintenance: A Machine Learning Approach. *J. Artif. Intell. Res.* 18 (2003), 149–181.
- [13] Bin Liu, Laura Chiticariu, Vivian Chu, HV Jagadish, and Frederick R. Reiss. 2010. Automatic rule refinement for information extraction. *Proc. VLDB Endowment* 3, 1-2 (2010), 588–597.
- [14] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. 2006. XML access control using static analysis. *ACM Trans. Inf. Syst. Secur.* 9, 3 (2006), 292–324.
- [15] Sudeepa Roy, Laura Chiticariu, Vitaly Feldman, Frederick R. Reiss, and Huaiyu Zhu. 2013. Provenance-based dictionary refinement in information extraction. In *2013 SIGMOD Int. Conf. Mgmt. of Data*. ACM, 457–468.
- [16] Sunita Sarawagi. 2008. Information Extraction. *Foundations and Trends in Databases* 1, 3 (2008), 261–377.
- [17] Latanya Sweeney. 1997. Weaving technology and policy together to maintain confidentiality. *J. of Law, Medicine & Ethics* 25, 2-3 (1997), 98–110.