ExPRESS – Extraction Pattern Recognition Engine and Specification Suite

Jakub Piskorski

Joint Research Center of the European Commission
Web Mining and Intelligence Action
Institute for the Protection and Security of the Citizen
Via Fermi 1, 21027 Ispra (VA), Italy

Abstract. The emergence of information extraction (IE) oriented pattern engines has been observed during the last decade. Most of them exploit heavily finite-state devices. This paper introduces ExPRESS – a new extraction pattern engine, whose rules are regular expressions over flat feature structures. The underlying pattern language is a blend of two previously introduced IE oriented pattern formalisms, namely, JAPE, used in the widely known GATE system, and the unification-based XTDL formalism used in SProUT. A brief and technical overview of ExPRESS, its pattern language and the pool of its native linguistic components is given. Furthermore, the implementation of the grammar interpreter is addressed too.

1 Introduction

The task of information extraction (IE) is centered around extracting specific structured information from free-text documents. The classical IE tasks focus on detecting entities, identifying relations which hold among them, and extracting events. Typically, the major step in the process of retrieving the sought-after information consists of applying a cascade of so called extraction patterns. Recently, the emergence of IE-oriented pattern specification languages has been observed. These languages utilize various types of formalisms, ranging from character-level regular expressions to unification-based formalisms. Due to efficiency reasons, finite-state based pattern engines are the most prominent ones being used.

This paper introduces ExPRESS (Extraction Pattern Recognition Engine and Specification Suite) – a new extraction pattern engine, whose rules are regular expressions over flat feature structures, i.e., non-recursive feature structures, where features are string valued. The rule specification language is a blend of two previously introduced IE-oriented grammar formalisms, namely, JAPE [1] used in the widely known GATE platform and the unification-based formalism XTDL deployed in SProUT [2]. The main motivation beyond the development of ExPRESS comes from: (a) a need of an efficient pattern engine for extracting facts from vast amount of news articles collected on a daily basis from the web

by Europe Media Monitor[1] (EMM) system [3], and (b) due to efficiency problems encountered when using other freely available IE-oriented pattern engines, including the two aforementioned ones.

The rest of this paper is organized as follows. We start in section 2 with some basic definitions and notions used throughout this paper. Next, in section 3 a brief overview of the related work is given. Subsequently, in section 4 ExPRESS, its pattern specification language and its core native linguistic components are described. Efficiency issues in the context of compiling and processing the grammars are addressed in section 5. Section 6 gives technical details about implementation and provides some figures concerning the run-time behavior. We provide a concluding summary in section 7.

2 Basic Definitions and Notions

This section introduces the basic definitions and notions used in this paper. A *deterministic finite-state automaton* (DFSA) is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite *set of states*, Σ is the *alphabet* of M, $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*, q_0 is the *initial state* and $F \subseteq Q$ is the *set of final states*. The transition function can be extended to $\delta^* : Q \times \Sigma^* \rightarrow Q \cup \{\bot\}$ by defining $\delta^*(q, \epsilon) = q$, $\delta^*(q, a) = \delta(q, a)$ if $\delta(q, a)$ is defined or $\delta^*(q, a) = \bot$ otherwise, and $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$ for $a \in \Sigma$ and $w \in \Sigma^*$. The *language accepted* by a DFSA M is defined as $L(M) = \{w \in \Sigma^* | \delta^*(q_0, w) \in F\}$. Languages accepted by finite-state automata are also called *regular*. The *union* and *concatenation* of two regular languages L_1 and L_2 is denoted as $L_1 \cup L_2$ and $L_1 \cdot L_2$ respectively. A *path* in a DFSA M is a sequence of triples $\langle (p_0, a_0, p_1), \ldots, (p_{k-1}, a_{k-1}, p_k) \rangle$, where $(p_{i-1}, a_{i-1}, p_i) \in Q \times \Sigma \times Q$ and $\delta(p_i, a_i) = p_{i+1}$ for $1 \leq i < k$. The string $a_0 a_1 \ldots a_k$ is the *label* of the path. Among all DFSAs recognizing the same language, there is always one which has the minimal number of states. We call such an automaton *minimal* (MDFSA). The definition of *nondeterministic finite-state automata* (NFSA) is analogous, with the difference that transition function is set-valued, i.e., more than one transition from a given state q labeled with a symbol $a \in \Sigma$ might exist.

Next, we define flat feature structures, which are frequently referred to in this paper. A type space is a triple $\Phi = (\Sigma_T, \Sigma_F, \Delta)$, where Σ_T is a finite *set of types*, Σ_F is a finite *set of features* and $\Delta : \Sigma_T \rightarrow 2^{\Sigma_F}$ is the total *type specification function*, i.e., Δ maps types to their features. We say that a feature $f \in \Sigma_F$ is *appropriate* for the type α if $f \in \Delta(\alpha)$, otherwise f is *inappropriate* for the type α. A *flat feature structure* (FFS) in the type space $\Phi = (\Sigma_T, \Sigma_F, \Delta)$ is a pair $s = (\alpha, val)$, where $\alpha \in \Sigma_T$ (α is a type), and $val : \Delta(\alpha) \rightarrow {\Sigma^+}_V \cup \{\top\}$ is a *feature-value mapping*, where Σ_V is a finite set of symbols. The symbol $\top$ is used to denote unspecified (undefined) feature values, i.e., $val_s(f) = \top$ means that the value of f is unspecified for s. We say, that two FFSs $s = (\alpha_s, val_s)$ and $t = (\alpha_t, val_t)$ *match* in the type space Φ if and only if: (a) $\alpha_s, \alpha_t \in \Sigma_T$,

[1] http://emm.jrc.it/overview.html

(b) $\alpha_s = \alpha_t$, and (c) $\forall f \in \Delta(\alpha_s) : val_s(f) = val_t(f)$ or $val_s(f) = \top$ or $val_t(f) = \top$. For the sake of simplicity, we denote a FFS $s = (\alpha, val)$ also as $[f_1 : v_1 \ldots f_k : v_k]_\alpha$, where $\forall 1 \leq i \leq k : f_i$ is appropriate for α and $v_i = val(f_i)$.

In this paper, we also refer to *typed feature structures* (TFS), which are related to record structures in programming languages and are widely used as a data structure for NLP. Their formalizations [4] include multiple inheritance and subtyping, which allow for terser descriptions.

3 Related Work

The idea of using regular expressions over more complex structures is not new and has been considered by several authors, e.g., [5] uses regular grammars with predicates over morphologically analyzed tokens. Furthermore, [6] introduces finite-state transducers with arbitrary predicates over symbols and discusses various operations on such finite-state devices. In particular, during the last decade, several high-level IE-oriented specification languages for creating patterns have been developed, e.g., [7] introduced CPSL designed as a language for specifying finite-state grammars over arbitrary annotations. The widely-known GATE platform, exploited heavily for development of IE components, comes with JAPE – Java Annotation Pattern Engine [1], which is similar in spirit to CPSL. A JAPE grammar consists of pattern-action rules. The left-hand side (LHS) of a rule is a regular expression over arbitrary atomic feature-value constraints, while the right-hand side (RHS) constitutes a so-called *annotation manipulation statement* which specifies the output structures to be produced once the pattern matches. Additionally, the RHS may call native code, which on the one hand provides a gateway to the outer world, but on the other hand makes pattern writing difficult for non-programmers.

A somewhat more declarative and linguistically-oriented pattern specification formalism called XTDL is used in SProUT [2], a lesser known IE framework. It can be seen as an amalgam of finite-state and unification-based grammar formalisms. In XTDL the LHS of a rule is a regular expression over typed feature structures (TFS) with functional operators and coreferences[2], and the RHS is a TFS, specifying the output production. Functional operators are primarily utilized for forming the slot values in the output structures and, secondly, they can act as Boolean-valued predicates, which allows for introducing complex constraints in the rules. The aforementioned features make XTDL more amenable formalism than JAPE since writing 'native code' is eliminated and coreferencing allows for terser descriptions.

Clearly, rich annotations on automata edges allow for compact descriptions, but standard finite-state optimization and processing methods are hardly applicable. Although, efficient processing techniques for both JAPE [8] and XTDL [9] have been developed, to the authors knowledge and experience processing even

[2] Coreferences express structural identity, create dynamic value assignments, and serve as means of data transfer from LHS to RHS of a pattern

moderate-size grammars with the aforementioned engines remains a bottle-neck.[3] In particular, processing XTDL patterns involves unification, a rather expensive operation.

Some other IE-oriented pattern languages are surveyed in [10], but since most of them are bound to a specific type of information and exhibit somewhat black-box character, we do not discuss them any further.

4 ExPRESS

4.1 Overview

ExPRESS is a pattern engine which allows for specifying and processing cascaded finite-state grammars, where grammar rules are regular expressions over feature structures. It has been mainly designed for tackling IE tasks. ExPRESS consists basically of a grammar parser and a cascaded-grammar interpreter. A cascaded grammar specification is divided into three parts: (a) *types declaration*, (b) a set of *grammar definitions* and (c) a *workflow specification*. The *types declaration* part is a list of all types and appropriate features for these types, which are used in the grammar(s). In the type declaration example in figure 1, three types are introduced, namely `person`, `person_group` and `violent_event`, where for each of them a list of appropriate features is specified.

```
person:=[NAME,FIRST_NAME,LAST_NAME,INITIAL,AMOUNT,SEX]
person_group:=[NAME,AMOUNT,QUANTIFIER]
violent_event:=[TYPE,METHOD,ACTOR,VICTIM]
```

Fig. 1. Type declaration in ExPRESS

A single *grammar definition* consists of two parts: a *grammar configuration* part and a *rule definition* part. In the configuration part, a list of arbitrary processing resources can be specified, which will be applied before the interpreter applies the grammar. These components provide the grammar interpreter with a stream of input flat feature structures represented as a list of disjunctions of FFSs. The list of available components and the task of integration of external components is addressed in section 4.3. Further, for each grammar a different search strategy can be chosen. Currently the following strategies are supported: (a) *longest-match*, (b) *all-matches*, and (c) *all-longest-matches* (*longest-match* strategy applied at each position in the input). Finally, the last item in the configuration part specifies the output production option. Three alternative options

[3] There are several implementations of JAPE. We did not test the recently developed Japec version [8], which is supposed to be 2-5 times faster than the original implementation.

are provided: (a) return only structures produced via grammar application, (b) additionally to (a) return also feature structures produced by other processing modules applied at the same level, (c) like (b), with the difference that only those feature structures produced by other processing modules are returned, which were not consumed by the application of the grammar. The simplified example in figure 2 gives an idea of the syntax of a single grammar. The rule specification format is described in detail in section 4.2.

```
SETTINGS:
 { MODULES: <Tokenizer>, <Morphology>, <Gazetteer>
   SEARCH_MODE: longest_match
   OUTPUT: grammar_only
 }
RULES:
 R1
 .
 .
 RN
```

Fig. 2. Syntax of a single grammar

Finally, the last part of the input to the parser, namely *workflow specification*, is a sequence of grammar names, which defines the order in which the grammars are applied by the interpreter. In addition, each grammar name, may be accompanied by a file, which specifies the priorities for the rules in the corresponding grammar. Thus, experimenting with different prioritization set-ups is more elegant than in JAPE, where priorities are encoded directly in the rules (XTDL also separates the prioritization settings from the grammars). If there are several rules which match and have the same priority, then all output structures are returned by the grammar interpreter, unless the output structures are identical. In the latter case only one instance is returned.

4.2 Rule Specification Language

This subsection focuses on the particularities of the rule specification formalism of ExPRESS, which is similar in spirit to JAPE, but also encompasses some features and syntax borrowed from XTDL. The LHS of a rule is a regular expression over flat feature structures (FFS), i.e., non-recursive TFS without coreferencing, where features are string-valued and unlike in XTDL types are not ordered in a hierarchy (see 2). On the LHS of a rule variables can be tailored to the string-valued attributes in order to facilitate information transport into the RHS, etc. Further, like in XTDL, functional operators are allowed on the RHSs for manipulating slot values and for establishing contact with the 'outer world'. They can also be deployed as boolean-valued predicates. There is

a predefined set of available functional operators, and new ones can be added by simply implementing an appropriate programming interface by the grammar developer. Finally, we adapted the JAPE's feature of associating patterns on the LHSs with multiple actions (*labeling*), i.e., producing more than one annotation (eventually nested) for a given text fragment.[4] A rule for matching person names presented in figure 3 illustrates the syntax. This rule matches a sequence

```
person  :> ((dictionary & [TYPE: "first_name",
                           SURFACE: #first])
            (dictionary & [TYPE: "initial",
                           SURFACE: #in]
             token & [SURFACE: "."]) ?
            (token & [TYPE: "firstCapital",
                      SURFACE: #last])):name
-> name: person & [NAME: #full_name,
                   FIRST_NAME: #first,
                   LAST_NAME: #last,
                   INITIAL: #in
                   AMOUNT: "1"]
                & #full_name := ConcWithBlanks(#first,#in,#last)
                & ValidatePersonName(#full_name).
```

Fig. 3. A rule for recognition of person names

consisting of: a structure of type `dictionary` (output of the dictionary look-up tool) representing the first name, followed by an optional initial (a sequence of `dictionary` and `token` structures), and another structure representing a capitalized token (last name). The symbol `&` links a type name of the FFS with a list of feature-value pairs representing the constraints which have to be fulfilled. It should not be confused with the same symbol denoting unification in XTDL. The symbols `#first`, `#in` and `#last` establish variable bindings to the surface forms of the matched text fragments. Further, the label `name` on the LHS specifies the start/end position of the action defined on the RHS of the rule. This action produce a structure of type `person`, where the value of the slots `FIRST_NAME` , `LAST_NAME` and `INITIAL` is created via accessing the variables `#first`, `#in` and `#last` resp. The value of the `NAME` slot is computed via a call to a functional operator `ConcWithBlanks()` which concatenates its arguments and inserts a space character between them. Finally, the RHS contains a call to a functional operator `ValidatePersonName()` which acts as a boolean predicate and contacts some

[4] XTDL allows only for producing single output structures and does not provide the labeling facility, i.e., output structure correspond to the entire text fragment matched by the LHS pattern. However, there is a 'dirty' workaround consisting of accessing positional information of single feature structures matched by the LHS and using such information for redefining start/end position of the output structure.

external mechanism (i.e., morphological person name filtering) which estimates whether the current name is likely to be a person name or not, and returns an appropriate value. It is important to note that in order for a rule to match, all boolean-valued predicates in the RHS of the rule must hold.

```
killing_event :> ((person_group & [NAME: #n1,
                                   AMOUNT: #a1,
                                   QUANTIFIER: #q1]
                  | person & [NAME: #n1,
                              AMOUNT: #a1]):victim

                 (dictionary & [TYPE: "death_trigger",
                                FORM: "passive"
                                METHOD: #m])

                 (person_group & [NAME: #n2,
                                  AMOUNT: #a2,
                                  QUANTIFIER: #q2]
                  | person & [NAME: #n2,
                              AMOUNT: #a2]):killer
                ):event
-> killer: actor & [NAME: #n2,
                    AMOUNT: #a2,
                    QUANTIFIER: #q2],
   victim: dead & [NAME: #n1,
                   AMOUNT: #a1,
                   QUANTIFIER: #q1]
                & IsNonZeroQuantifier(#q1),
   event: violent_event & [TYPE: "killing",
                           METHOD: #m,
                           ACTOR: #n2,
                           VICTIM: #n1].
```

Fig. 4. A rule for violent event recognition

Another example of a rule that matches information concerning actors and victims in violent events, where a person or a group thereof is killed by another human body, is given in figure 4. This rule matches a sequence consisting of: a FFSs of type `person` or `person_group` (the disjunction is denoted with '`|`') representing a human(s) who is (are) the *victim* of the event, followed by a phrase in passive form, which triggers a 'killing' event (dictionary look-up), and another structure representing the *actor* (person or group of persons). There are three labels on the LHS, namely `victim`, `killer`, and `event`, which produce structures of type `dead`, `actor` and `violent_event` respectively. In case of the `dead` structure, the quantifier (variable `#q1`) must not be a 'zero' quantifier. This con-

```
dead & [NAME: "Talibani", AMOUNT: "230", QUANTIFIER: "Most of"]
actor & [NAME: "US troops"]
violent_event & [TYPE: "killing",
                 METHOD: "shooting",
                 ACTOR: "US troops",
                 VICTIM: "Talibani"].
```

Fig. 5. The output structures produced by the rule in figure 4 when matching the text fragment *Most of the 230 Talibani were shot by the US troops*

straint is expressed in the rule via the boolean predicate `IsNonZeroQuantifier`. The rule described above matches the text fragment *Most of the 230 Talibani were shot by the US troops* and produces three output structures depicted in figure 5. On the contrary, the text fragment *None of the Taliban were killed by UN troops* would not be matched since `IsNonZeroQuantifier` predicate ("*None of*") does not hold.

The handling of Kleene constructions has to be clarified briefly. If a structure containing a variable within a Kleene construction is matched more than once, then (optionally) a local instances of the variable is created for each such submatch, and the local bindings are accumulated into a concatenation thereof. This resembles the weak unidirectional coreferences in XTDL [9]. Further, labels are not allowed within Kleene constructions, and labeled construction are not allowed to consume empty input streams.

The full syntax of ExPRESS extraction rule formalism is given in BNF format in figure 6. Some constructs known from other pattern languages are missing, e.g., negation, but it can be simulated via non-productive rules and prioritization [1].

4.3 Native and External Linguistic Components

In order to facilitate writing grammars ExPRESS comes with a pool of native basic Unicode-aware IE-oriented linguistic processing resources, which includes: (a) a basic tokenizer which segments text based on a list of white spaces and token separators, (b) a tokenizer which additionally performs fine-grained token classification (circa 40 IE-oriented default token classes are provided, e.g. email addresses, URLs, hyphenated constructions, etc.), (c) simple morphological analyzer based on full-form lexica encoded in the MULTEXT[5] format [11], and (d) a space and time efficient dictionary look-up tool which allows for storing huge

[5] MULTEXT was a EU-funded project aiming at developing a set of generally usable software tools to manipulate and analyze text corpora, together with lexicons and multilingual corpora in several European languages. In particular, harmonized specifications for encoding computational lexicons have been established, i.e., same tagset and features are used for all languages.

```
Rules        -> Rule (Rule)*
Rule         -> RuleName ":=" Pattern "->" (Actions)? "."
RuleName     -> Identifier

Pattern      -> "(" Concat ")" (":" Label)?
Label        -> Identifier
Concat       -> Disjunction (Disjunction)*
Disjunction  -> Kleene ("|" Kleene )*
Kleene       -> Element ("+" | "*" | "?")?
Element      -> (BasicElement | Pattern)
BasicElement -> Type ("&" FeatStruct)?
Type         -> Identifier
FeatStruct   -> "[" Attribute ":" Value ("," Attribute ":" Value)* "]"
Attribute    -> Identifier
Value        -> (SimpleValue (Variable)?) | (Variable)
SimpleValue  -> Identifier
Variable     -> "#"Identifier

Actions      -> Action ("," Action)*
Action       -> Label ":" Type ("&" OutputStruct ("&" FuncOp)* )?
OutputStruct -> "[" Attribute ":" OVal ("," Attribute ":" OVal)* "]"
Attribute    -> Identifier
OVal         -> (SimpleValue | Variable)?
FuncOp       -> (Variable ":=")? FuncOpName "(" Arg ("," Arg)* ")"
FuncOpName   -> Identifier
Arg          -> (SimpleValue | Variable)
```

Fig. 6. ExPRESS Syntax

amount of entries, where each of them can be associated with arbitrary feature-value pairs. The latter two components exploit the finite-state compression and compilation techniques described in [12], [13] and [14].

Additional external processing components can be easily integrated via implementing a special programming interface. Basically this boils down to providing a function which converts components specific native output format into a stream of disjunctions of FFSs with positional information, and providing functions which return a list of types of output structures returned by this component and features which are appropriate for these types. The latter ones are utilized for performing a strict compatibility check with the types declared in the grammar cascade.

5 Compiling and Processing Grammars

Since the reservoir of FFSs used in extraction rules is potentially infinite, converting ExPRESS grammars into a single and optimized for processing finite-state network is not straightforward. Typically, in a grammar consisting of regular patterns over some feature structures the latter ones are replaced by some unique symbols representing references to these feature structures, i.e., they are treated in a symbolic way (naive implementation). Subsequently, single extraction patterns are merged into a single MDFSA via application of standard finite-state optimization techniques. Although such finite-state device is deterministic in a strict sense, it clearly is not deterministic when we consider the real semantics of its transition labels, i.e., feature structures. Consequently, while processing such automata (being the result of merging the elementary rule automata into one MDFSA), in each step, all outgoing transition from a given state are inspected one by one whether their label matches with the current input feature structures. Since distinct feature structures (even pairs of matching feature structures) are represented as different symbols, some states of the automaton, might have a quite high number of outgoing transitions. This applies in particular for the initial state and in its direct proximity. Inspecting all outgoing transition each time the initial state is visited clearly deteriorates the run-time performance.

The rest of this section describes a method for efficiently processing ExPRESS grammars. First, in in subsection 5.1, the pattern matching algorithm sketched above is described in a more formal manner. Next, in subsection 5.2, some enhancements thereof are introduced, which mainly consist of flattening FFSs in the patterns and input FFSs into character-level regular expressions and strings respectively, so that matching input FFSs with the grammar automaton can be performed efficiently.

5.1 Pattern Matching Algorithm

Let G be a grammar consisting of regular patterns $r_1 \ldots r_n$ over FFSs, where each pattern r_i is represented by a regular expression R_i. FFSs are replaced

in each R_i by symbols representing references to these FFSs. Next, we construct a DFSA M (representing the whole grammar) which accepts the language $R_1 \cdot \{\$_1\} \cup \ldots \cup R_n \cdot \{\$_n\}$, where $\$_1 \ldots \$_n$ are unique symbols representing rule identifiers. Additionally, we turn each state q into a final state if it has an outgoing transition labeled with one of the symbols in $\{\$_1, \ldots, \$_n\}$. All other states are non-final. Further, let us assume, that the stream of input FFSs is represented as a directed labeled graph $InputFS = (V, E)$, where all nodes in V correspond to start/end positions of text spans associated with the input FFSs. An edge in E is a 3-tuple (v, a, u), where v and u are source/target nodes, and a is the label which points to some FFS.

An algorithm that takes automaton M and finds all matches in $InputFS$ (an input stream of flat feature structures) is presented in figure 7. Please note that M is is not deterministic when we consider the real semantics of its transition labels. The variable *node* (initialized in line 1) points to the current node in $InputFS$, i.e., the node from which the algorithm tries to find the next potential match. The main **while** loop of the algorithm (lines 3-20) is executed until the current node is the last node in $InputFS$. Since there is potentially more than one path from the node u in $InputFS$ which matches with the automaton M and due to the fact that even one single path in $InputFS$ might match with different paths in M, we store in the set *Active* all 'current' configurations of M. A single *configuration* of M is a triple (q, π, v), where q denotes the current state of M, π is a sequence of input FFSs which match a path in M from q_0 to q, and v denotes the next node in $InputFS$ from which subsequent matches in the input stream will be sought. Analogously, in *Accepting* we store all *accepting configurations* of M (ones whose current state is final). Initially this set is empty (line 5). In the **while** loop in lines 6-15 all possible configurations of M that match some path in $InputFS$ starting in the node *node* are computed. This process resembles breadth-first-search in graphs. In particular, in the inner loop (lines 8-14) for each $(q, \pi, v) \in Active$ we compute all 'subsequent' configurations, i.e, the ones being the result of matching some input FFS a starting in node v with a FFS a' in the set of transitions for state q, so that $\delta(q, a') \neq \perp$. Matching test is done via a call to the function MATCHES (line 13). Note that for a single input FFS there might be potentially more than one matching transition in M (**for** loop in lines 12-13). Once all 'new' configurations have been computed, we select from the set of accepting configurations one which fulfills selection criteria (line 17). Selection criteria may vary, depending on the search strategy. For instance, in the *longest-match strategy*, one simply takes the configuration which covers the longest text span. If more than one such configuration exists, then the one being a result of application of a rule with highest priority is chosen, etc.[6] Once an accepting configuration is chosen, an appropriate action is performed (line 18), e.g., output structure(s) is produced. We can restore the rules that matched via inspecting transition labels from final states. Finally, the value of the current

[6] In some applications, it is convenient to select more than one accepting configuration, but the modification to the presented algorithm is straightforward so it is not discussed any further.

node in the input graph is then modified accordingly in the line 19. If no accepting configurations were found, the current node is set to the closest node in $InputFS$ that has an outgoing edge (line 20).

```
FIND-MATCHES(M = (Q, Σ, δ, q0, F), InputFS)
 1  node ← GETFIRSTNODE(InputFS)
 2  lastNode ← GETLASTNODE(InputFS)
 3  while node ≠ lastNode
 4  do Active ← {(q0, ϵ, node)}
 5     Accepting ← ∅
 6     while Active ≠ ∅
 7     do Next ← ∅
 8        for (q, π, v) ∈ Active
 9        do if q ∈ F
10              then Accepting ← Accepting ∪ {(q, π, v)}
11           for (v, a, u) ∈ InputFS
12           do for a′ ∈ Σ : δ(q, a′) ≠ ⊥
13              do if MATCHES(a, a′)
14                    then Next ← Next ∪ {(δ(q, a′), π · a′, u)}
15        Active ← Next
16     if Accepting ≠ ∅
17       then (q, π, v) ← SELECTACCEPTINGCONFIG(Accepting)
18            EXECUTEACTION(M, q, π)
19            node ← v
20       else node ← GETNEXTNODE(InputFS, node)
21  return
```

Fig. 7. Pattern matching algorithm

Intuitively, the most time-consuming part of the algorithm in figure 7 is the **for** loop in lines 12-14. In the naive implementation one has to inspect all outgoing transitions from the state q whether their label (a') matches with the current input FFS (a). Inspecting all outgoing transition for frequently visited states, e.g., the initial state and its direct proximity, clearly deteriorates the run-time performance.

For alleviating the aforementioned problem JAPE applies a solution which exploits the fact that the feature structures being labels of outgoing transitions from a given state have shared parts. In particular, all such structures are partitioned into disjoint partial feature structures which do not intersect and they are reordered accordingly in order to avoid redundant computations while matching the stream of input feature structures.

In XTDL, where the recognition part of the rules consists of TFSs, a similar technique for ordering the outgoing transitions is used. It consists of comput-

ing a transition hierarchy under TFS subsumption for all outgoing transitions (labels) of a given state. While traversing the grammar automaton, these transition hierarchies are utilized for inspecting outgoing transitions from a given state, starting with the least specific transition(s) first, and moving downwards in the hierarchy, if necessary. Although this technique proved to give a significant speed-up, the number of transitions which have to be inspected for computing 'subsequent' automaton configurations might be on an average relatively high due to the low degree of feature-value sharing.

5.2 Matching Flat Feature Structures

In order to efficiently perform the crucial matching step in the algorithm described in the previous section (lines 12-14) we apply in ExPRESS a technique which consists of flattening input FFSs into strings and converting all transitions labels of a given state into a single DFSA, so that computing 'new' target states (new automata configurations) is reduced to performing a simple deterministic automaton look-up.

Generally speaking, the process of finding a match at a given position in the input stream is split into three steps: (1) selection of the sequence(s) of input FFSs which is (are) covered by some rule(s) according to predefined selection strategy, (2) performing a fully-fledged match of the selected rule(s) against the selected input sequence of FFSs, which includes variable and label binding, and (3) producing and merging output structures. Postponing variable and label binding allows for efficiently implementing step (1). Further, once an input sequence and the rules (or more) that match this sequence have been selected, performing full matching in step (2) can be done quickly due to the limited number of applicable rules. Thus, step (1) can be seen as a prefiltering of applicable rules. Since there are potentially several paths in the automaton for the rule(s) selected in step (2), step (3) is necessary for merging and/or filtering out some output structures, but we do not describe it here any further.

We now turn to implementing step (1) and sketch the technique for quick computing matching transitions from a given state in a semi-formal way. Firstly, let us observe that only a finite number of feature-value pairs are used in the grammar rules. We can compute for all FFSs of a given type α, which appear in the rules, the respective value sets $\Sigma_1, \ldots \Sigma_k$, where Σ_i is the value-set for the i-th feature appropriate for the type α.[7] A given input FFS $s = [f_1 : v_1 \ldots f_k : v_k]_\alpha$ can be then encoded as a string $id(\alpha)\cdot\$\cdot{v^*}_1\cdot\$\ldots\$\cdot{v^*}_k$, where id maps types to unique symbols representing their identifiers, \$ is a unique symbol $\notin \Sigma_i \cup \{\top\}$ ($\forall 1 \leq i \leq k$) which represents a separator and ${v^*}_i \in \Sigma_i \cup \{\top\}$ are defined as follows:

$$
{v^*}_i = \begin{cases} v_i : v_i \in \Sigma_i \\ \top : v_i \notin \Sigma_i \ \vee v_i = \top \end{cases}
$$

For instance, an input FFS $[pos : noun, case : loc, gen : fem]_{morph}$ with pos, $case$, and gen being appropriate features for the type $morph$, where $\Sigma_{pos} =$

[7] Note that we order the features appropriate for a given type

$\{noun, adj\}$, $\Sigma_{gen} = \{masc, fem\}$, and $\Sigma_{case} = \{nom, acc, dat, gen\}$ (seen feature values), would be represented as the following string: $id(morph) \cdot \$ \cdot noun \cdot \$ \cdot \top \cdot \$ \cdot fem$.

Analogously, each FFS $s = [f_1 : v_1 \ldots f_k : v_k]_\alpha$ being a label of a transition t from a given state in the grammar automaton is represented as a regular expression of the form $id(\alpha) \cdot \$\ v^*{}_1 \cdot \$ \ldots \$ \cdot v^*{}_k \cdot \% \cdot trans(t)$, where id and \$ are defined as previously, % is another unique separator, *trans* maps transitions to their unique symbolic identifiers, and $v^*{}_i \in (\Sigma_i \cup \{\top\})^*$ is a regular expression defined as follows:

$$v^*{}_i = \begin{cases} v_i & : v_i \in \Sigma_i \\ \{\top\} \cup \Sigma_i & : v_i = \top \end{cases}$$

The second part of the definition of $v^*{}_i$ has to be a disjunction of $\{\top\}$ and Σ_i since we intend to merge all regular expressions representing transitions from a given state into a single DFSA ('transition' automaton for a given state), i.e., in case of encoding a feature with unspecified value, all values (for that feature and type) seen in other patterns have to be considered (Σ_i). Now, let $T_1, \ldots, T_n$ be the regular expressions representing the labels of the transitions $t_1, \ldots, t_n$ from a given state q in M resp., which were obtained in the previously described manner. Let M_q be a DFSA which accepts the language $T_1 \cup \ldots \cup T_n$. Then, we can compute the set of possible target states for the state q in M and an input FFS a that is represented as a string w simply via computing a target state $p = \delta_{M_q}(q, w)$ in M_q and inspecting all outgoing paths from p, whose labels start with % in order to retrieve the target state identifiers in the grammar automaton M. In this way, the steps 12-14 in the algorithm in figure 7 are reduced to a simple string matching with the DFSA M_q.

We give an example to clarify the aforementioned technique. Let us assume that t_1 and t_2 are two outgoing transitions from state q, which are labeled with $[pos : noun, case : \top, gen : \top]_{morph}$ and $[pos : \top, case : acc, gen : \top]_{morph}$ and which lead to state q_1 and q_2 resp. Turning them into corresponding regular expressions yields $id(morph) \cdot \$ \cdot noun \cdot \$ \cdot \{nom, acc, gen, dat, \top\} \cdot \$ \cdot \{fem, masc, \top\} \cdot \%q_1$ for t_1 and analogously $id(morph) \cdot \$ \cdot \{noun, adj, \top\} \cdot \$ \cdot acc \cdot \$ \cdot \{fem, masc, \top\} \cdot \%q_2$ for t_2. The result of merging regular expressions representing the labels of t_1 and t_2 into one DFSA M_q is shown in figure 8 in a simplified form (\$ symbols were omitted).

Let us assume that an input FFS $s = [pos : noun, case : acc, gen : masc]_{morph}$ has to be matched against the grammar automaton M in state q. Matching the string representation of s, i.e., $id(morph) \cdot \$ \cdot noun \cdot \$ \cdot acc \cdot \$ \cdot masc$, in the transition automaton M_q results in state 8. Consequently, both states q_1 and q_2 are reachable via matching FFS s in M from state q.

Techniques similar to the one described in this section are also used in other finite-state based frameworks, e.g., in [15]. A further improvement could be achieved by turning all input FFSs at a given position into a union of their corresponding string representations and subsequently performing on-the-fly intersection thereof with the 'transition' automaton representing the outgoing transitions from a given state. Whether this results in an enhanced run-time performance is

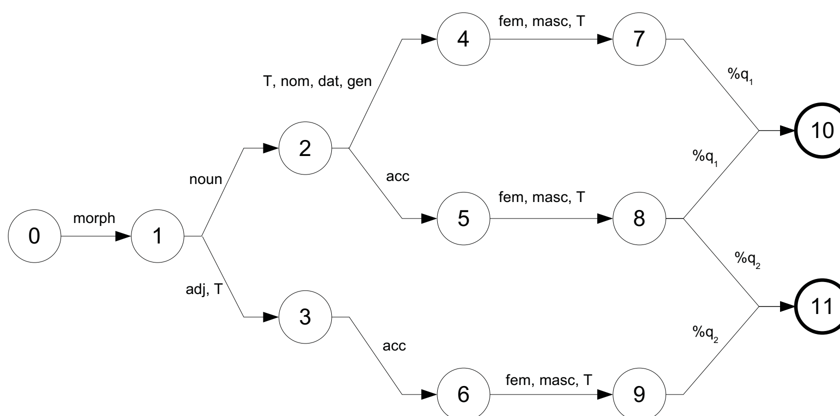

Fig. 8. Transitions labels merged into a single DFSA

unclear since intersection operation is more time-consuming than a single string acceptance check.

6 Technicalities

ExPRESS has been implemented fully in JAVA. The development is based on the Java Compiler Compiler [16] and the Java package `dk.brics.automaton` containing time efficient implementations of finite-state automata and a bag of standard operations for manipulating and optimizing them [17]. Currently, ExPRESS consists of two stand-alone programs (parser and interpreter) and a documented JAVA API for facilitating integration into other frameworks. Making ExPRESS publicly available for research purposes is envisaged at a later stage.

We have carried out some experiments to measure the run-time behavior of Express with a two-stage grammar for recognition of information on actors, kidnapped, dead and wounded in violent events. In the first stage standard named entities are recognized, e.g., persons, group of persons, numerical expressions, etc. In the second stage, single-slot and two-slot extraction rules are applied to retrieve the sought-after information on related events, in which the entities recognized in the first stage participate. The first-stage grammar consisting of circa 100 rules was developed by an expert, whereas the second-level grammar was obtained via semi-automatic conversion of ca. 3000 automatically learned IE patterns [18] into ExPRESS rules. Further, five linguistic processing resources (e.g., tokenizer, gazetteer and morphology look-up) were involved in the extraction process. Subsequently, the aforementioned two-level grammar has been converted in almost one-to-one manner into a XTDL grammar. It turned to be

a relatively simple task since the core linguistic components provided with ExPRESS have nearly identical functionality and I/O specification as those used in SProUT. However, some rules had to be expressed as two rules in XTDL since XTDL rules do not allow for specifying more than one output structure directly.

In an experiment, the grammars were applied to a 167 MB excerpt of the news in English on terrorism, consisting of 122 files on a PC Pentium 4 machine with 2,79 GHz. The table 1 gives figures of the average run-time (in seconds) for processing a single file (average size of 1,37 MB) at different stages. The average number of matches per document amounted to ca. 60 000. Clearly, ExPRESS

Time \ Grammar Interpreter	XTDL	ExPRESS
core linguistic components stage I	2.451	1.818
entity-pattern matching	38.212	1.923
entity-structure production	4.172	0.515
core linguistic components stage II	1.092	0.639
event-pattern matching	12.124	0.666
event-structure production	0.156	0.013
Total	58.207	5.574

Table 1. Run-time behavior (in seconds): XTDL vs. ExPRESS

performs significantly better than XTDL interpreter. The pattern matching itself constituted 46, 34% (ExPRESS) and 86, 48% (XTDL) of the total processing time respectively. In a second experiment, we have slightly 'compressed' the XTDL grammar through using coreferencing and other XTDL specific features, which resulted in deterioration of the run-time performance by the factor of two.

Finally, in the last experiment, we applied the same cascade of grammars to a collection of sentences (8 MB), where for each sentence in this collection there is at least one second-stage extraction rule that matches. ExPRESS run-time amounted to 36,7 seconds, whereas SProUT needed for processing the same collection ca 575 seconds.

Although converting ExPRESS grammars into JAPE format is a more laborious task, the above run-time figures for ExPRESS are better than one could potentially obtain when using JAPE according to the author's 'subjective' experience with the latter one and some basic experiments of converting the first-level grammar into JAPE.

7 Summary

In this paper, we presented ExPRESS, a new IE-oriented pattern specification and recognition engine, which borrows heavily from two previously introduced

pattern languages, namely JAPE and XTDL. In particular, ExPRESS grammars consist of extraction rules which are regular expressions over flat feature structures with string-valued features. ExPRESS was developed primarily in order to find a trade-off between 'compact descriptions' and efficient processing of huge text collections. It is already operational and it is being deployed in a real-time news event extraction system for detecting violent and natural disaster events [19]. In particular, ExPRESS is capable of applying modest-size event extraction grammars on MB-sized texts within seconds. Clearly, XTDL or some other IE-oriented pattern languages are more expressive and more powerful, but there is a wide range of extraction tasks for which ExPRESS will come in handy and might constitute a time-efficient alternative.

In future work, the pattern formalism will be extended by adding some new constructs and providing new native processing resources. Going beyond 'sequential' processing of grammars is planned. In general, ExPRESS will be kept as minimal as possible and any future developments will be strictly driven by the needs of specific applications. In particular, it will be deployed for named-entity and relation extraction. Finally, exploring additional performance enhancing techniques for processing grammars is envisaged, e.g., (a) intelligent reordering of feature-value pairs in the FFS in such a way that features which are most likely to eliminate a high number of potential target states precede other feature-value pairs, and (b) conversion of input FFSs starting at a given position into a union of their corresponding string representations and subsequently performing on-the-fly intersection thereof with the 'transition' automaton representing the outgoing transitions from a given state.

Acknowledgments

The work presented in this paper was supported by the Europe Media Monitoring Project (EMM) carried out by the Web Mining and Intelligence Action in the Joint Research Centre of the European Commission. I would like to thank Hristo Tanev, Vanni Zavarella, Bruno Pouliquen, Ralf Steinberger, Camelia Ignat and other EMM colleagues for fruitful discussions. Finally, I am indebted to Clive Best and Delilah Al-Khudhairy for valuable comments and proof-reading of this paper.

References

1. Cunningham, H., Maynard, D., Tablan, V.: JAPE: a Java Annotation Patterns Engine (Second Edition). Technical Report, CS–00–10, University of Sheffield, Department of Computer Science (2000)
2. Drożdżyński, W., Krieger, H.U., Piskorski, J., Schäfer, U., Xu, F.: Shallow Processing with Unification and Typed Feature Structures — Foundations and Applications. Künstliche Intelligenz **2004(1)** (2004) 17–23
3. Best, C., van der Goot, E., Blackler, K., Garcia, T., Horby, D.: Europe Media Monitor. Technical Report EUR 22173 EN, European Commission. (2005)

4. Copestake, A.: Appendix: definitions of typed feature structures. Natural Language Engineering **6**(1) (2000) 109–112
5. Neumann, G., Backofen, R., Baur, J., Becker, M., Braun, C.: An information extraction core system for real world German text processing. In: Proceedings of the 5th International Conference of Applied Natural Language. (1997) 208–215
6. van Noord, G., Gerdemann, D.: Finite State Transducers with Predicates and Identity. *Grammars* **4(3)** (2001) 263–286
7. Appelt, D., Onyshkevych, B.: The Common Pattern Specification Language. In: Proceedings of Tipster Text Program - Phase III. (1998) 23–30
8. Japec: Japec – A Jape-to-Java Optimizing Compiler. Web document, http://www.ontotext.com/gate/JapecPres.pdf (2006)
9. Drożdżyński, W., Krieger, H.U., Piskorski, J., Schäfer, U., Xu, F.: A Bag of Useful Techniques for Unification-Based Finite-State Transducers. In: Proceedings of of 7th KONVENS Conference, Vienna, Austria. (2004)
10. Muslea, I.: Extraction Patterns for Information Extraction Tasks: A Survey. In: Proceedings of AAAI 1999. (1999)
11. Erjavec, T.: MULTEXT - East Morphosyntactic Specifications (2004)
12. Daciuk, J.: Incremental Construction of Finite-State Automata and Transducers. PhD Thesis. Technical University Gdańsk. (1998)
13. Piskorski, J.: On Compact Storage Models for Gazetteers. In: Proceedings of the 5th International Workshop on Finite-State Methods and Natural Language Processing, Helisnki, Finland, Springer, LNAI (2005)
14. Daciuk, J., Piskorski, J.: Gazetteer Compression Technique Based on Substructure Recognition. In: Proceedings of Intelligent Information Systems 2006 - New Trends in Intelligent Information Processing and Web Mining, Springer Verlag series "Advances in Soft Computing" (2006)
15. Skut, W., Ulrich, S., Hammervold, K.: A Flexible Rule Compiler for Speech Synthesis. In: Proceedings of the International IIS:IIP WM'2004 Conference. Zakopane, Poland. Springer, Advances in Soft Computing. (2004) 257–266
16. JavaCC: https://javacc.dev.java.net
17. Moller, A.: http://www.brics.dk/automaton (2007)
18. Piskorski, J., Tanev, H., Oezden-Wennerberg, P.: Extracting Violent Events from On-line News for Ontology Population. In: 10th International Conference on Business Information Systems. Poznan, Poland. Lecture Notes in Computer Science, LNCS 4439. (2007) 287–300
19. Tanev, H., Piskorski, J., Atkinson, M.: Real-Time News Event Extraction for Global Crisis Monitoring. In: Proceedings of the 13th International Conference on Applications of Natural Language to Information Systems (NLDB 2008), London, UK, 24–27 June, 2008. Lecture Notes in Computer Science Vol 5039, Springer Verlag Berlin Heidelberg. (2008) 207–218