Natural Language Engineering **22** (1): 1–40. © Cambridge University Press 2014
doi:10.1017/S1351324914000114

UIMA Ruta: Rapid development of rule-based information extraction applications

PETER KLUEGL[1], MARTIN TOEPFER[2], PHILIP-DANIEL BECK[2], GEORG FETTE[2] and FRANK PUPPE[2]

[1]*Comprehensive Heart Failure Center, University of Würzburg, Straubmühlweg 2a and Department of Computer Science VI, University of Würzburg, Am Hubland, Würzburg, Germany*
email: peter.kluegl@uni-wuerzburg.de

[2]*Department of Computer Science VI, University of Würzburg, Am Hubland, Würzburg, Germany*
email: {martin.toepfer, philip.beck, georg.fette, frank.puppe}@uni-wuerzburg.de

(*Received 4 Feburary 2014; revised 21 August 2014; accepted 26 August 2014; first published online 8 October 2014*)

Abstract

Rule-based information extraction is an important approach for processing the increasingly available amount of unstructured data. The manual creation of rule-based applications is a time-consuming and tedious task, which requires qualified knowledge engineers. The costs of this process can be reduced by providing a suitable rule language and extensive tooling support. This paper presents UIMA Ruta, a tool for rule-based information extraction and text processing applications. The system was designed with focus on rapid development. The rule language and its matching paradigm facilitate the quick specification of comprehensible extraction knowledge. They support a compact representation while still providing a high level of expressiveness. These advantages are supplemented by the development environment UIMA Ruta Workbench. It provides, in addition to extensive editing support, essential assistance for explanation of rule execution, introspection, automatic validation, and rule induction. UIMA Ruta is a useful tool for academia and industry due to its open source license. We compare UIMA Ruta to related rule-based systems especially concerning the compactness of the rule representation, the expressiveness, and the provided tooling support. The competitiveness of the runtime performance is shown in relation to a popular and freely-available system. A selection of case studies implemented with UIMA Ruta illustrates the usefulness of the system in real-world scenarios.

1 Introduction

Information extraction addresses the identification of well-defined entities and relations in unstructured data and especially in textual documents. Even if the research in this task has a long history, it becomes more and more important nowadays due to the increased availability of unstructured data. In order to access the concealed information for analytic processes, it has to be transformed

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

into a structured representation. Hence, information extraction has become a key component in the integration of textual data and can be considered as an umbrella term for many interesting tasks such as named entity recognition, sentiment analysis or knowledge extraction. Two emerging and challenging trends in this area are information extraction from social media, like blogs or tweets (Piskorski and Yangarber 2013), and knowledge extraction from clinical notes (Savova *et al.* 2010).

Approaches to information extraction can roughly be divided into two main categories: Approaches based on handcrafted rules and approaches based on statistical models trained in a supervised fashion. The latter models include classifiers or probabilistic graphical models like Conditional Random Fields (Lafferty, McCallum and Pereira 2001). There are of course no clear boundaries since hybrid information extraction systems can apply components of both approaches, or the rules are not written by a knowledge engineer, but they are automatically induced. While statistical models dominate the research in academia, commercial applications are mostly implemented as rule-based systems (Chiticariu, Li and Reiss 2013). This discrepancy cannot be explained by the latency of translational efforts from research to industry. Chiticariu *et al.* have investigated the reasons for this disconnect and noticed that research and industry measure the costs and benefits of information extraction differently (Chiticariu *et al.* 2013). Aside from many other reasons, rule-based systems sometimes fit better in the requirements of real-world use cases, e.g., availability of labeled data, stability of the specification, or traceability of results. Even though statistical models often perform better in information extraction challenges, the need for rule-based information extraction will not decrease in the foreseeable future. The amount of publications about rule-based systems and approaches has even been slightly increasing in the recent years (Doan *et al.* 2008). Approaches for rule-based information extraction provide also advantages in combination with statistical models. Rules can be applied for high-level feature extraction, for solving different pre- and postprocessing tasks, and for semi-automatic creation of gold standards. It is often faster to engineer one rule than to annotate repeating mentions of a specific entity.

Rule-based information extraction systems mostly consist of a specification of a text-based rule language and an interpreter, which is able to apply the rules on documents in order to identify new information (Appelt and Onyshkevych 1998). The textual representation of rules leads to a development process where a knowledge engineer manually writes rules. These rules are composed of a condition part and an action part. The condition of the rule is a pattern of properties, which need to be fulfilled by an interesting position in the document. The properties are normally represented as annotations. These annotations assign a specific type and possibly additional features to a span of text. Such features may be capitalization, part of speech, formatting, or presence in a dictionary. Since the sequential order of properties is very important for the specification of patterns, the condition part often represents a regular expression over annotations possibly extended with additional constraints. If the regular expression matches on a text position, then the action part modifies the annotations. In the majority of cases, new annotations are added, which represent interesting entities and relations, or complex properties.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

The matching algorithm of a set of rules (rule grammar) is often implemented as a Finite State Transducer (FST), an automaton which traverses the annotation lattices and creates or modifies annotations (Cunningham, Maynard and Tablan 2000; Drozdzynski *et al.* 2004; Boguraev and Neff 2010). The automaton processes the document just once and does not react on its own modifications. A popular strategy is the usage of cascaded rule grammars, where one grammar is based on the results of previous grammars. This approach provides many engineering advantages, for example, the easier specification of complex patterns by describing them as a combination of simpler ones, or the clear separation of the stages of engineering approaches and their contexts (Boguraev and Neff 2006).

The creation of rule-based information extraction applications is a knowledge engineering process and its success depends on diverse factors. In contrast to approaches based on machine learning techniques, the rules are normally written by a human knowledge engineer, whose availability and training have major influence on the quality of the application. Developing rule-based information extraction applications can be a time-consuming and tedious task and a qualified knowledge engineer is an expensive resource. Thus, she should be supported by a compact rule representation and extensive tooling in order to guarantee a rapid and efficient development process. The rule language itself needs to provide a level of expressiveness that facilitates the effective specification of the patterns and actions. Both factors are able to reduce development time and costs. Finally, applications may require a certain order of runtime performance, which influences other factors.

This paper presents UIMA Ruta (Rule-based Text Annotation)[1], a rule-based tool that focuses especially on the rapid development of information extraction and even general text processing applications. The system consists of a rule language extended with scripting elements and a strong development support. The rule language was designed to provide a compact and comprehensible representation of patterns over annotations without restricting its expressiveness or area of use. It covers almost all features of related rule languages for information extraction while still introducing a few new ones. Although the rule language was optimized for rapid development and not for runtime performance, it competes well with similar rule-based systems concerning speed. The UIMA Ruta Workbench provides a full-featured development environment for the UIMA Ruta language, which exceeds the functionality of most related tools. It was developed to ease every step in engineering rule-based applications and provides, in addition to classical editing support like syntax checking, various useful tools. The system can generate a complete explanation of each step of the rule inference, which enables the engineer to adapt rules responsible for unintended behavior. The engineering process is supported by tools for introspection, test-driven development, constraint-driven evaluation, and automatic rule induction. The combination of the provided tooling results in a rapid and agile development of well-maintained rule sets. UIMA Ruta is an open source project with an industry-friendly license and is now being

[1] An early version was published under the name TextMarker (Kluegl, Atzmueller and Puppe 2009d).

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

developed by an active community. The project page[2] contains further links to the documentation, examples, and tutorials.

The rest of the paper is structured as follows: Section 2 gives an overview of the related work in the area of rule-based information extraction systems. The rule-based scripting language is introduced in Section 3. The syntax, semantics, and the rule matching algorithm are explained in detail and illustrated with examples. Furthermore, a selection of special features and different engineering approaches complement the description of the language. Section 4 focuses on the development environment and available tooling for improving the engineering experience. In Section 5, a comparison of UIMA Ruta and related rule languages is provided. Section 6 gives a short overview of case studies using UIMA Ruta and Section 7 concludes with a summary.

2 Related work

The related work can be divided into two main categories: systems that provide a text-based rule language for information extraction tasks and development environments for supporting the engineering process. Rule-based information extraction is a well-studied field. We do not provide an exhaustive overview of related systems, but introduce only a selection of recent work. A comparison of UIMA Ruta to these systems is given in Section 5. An introduction to the historical development in this area can be found, for example, in Turmo, Ageno, and Català (2006). Before the related work is described, the main concepts of the architecture underlying UIMA Ruta are introduced as preliminaries for the next sections.

2.1 Architectures

Rule grammars for information extraction are often part of a larger pipeline of components. The rules build upon annotations added by the previous components most often for linguistic analysis of the document. These components include, for example, tokenizers, sentence detectors, gazetteers, part-of-speech taggers, morphological analyzers, or parsers. Rule-based systems are therefore integrated in an architecture or framework for natural language processing in the majority of cases. The architectures provide uniform models for data exchange, component interfaces and process control, which simplify the interoperability of the components and facilitate the specification of complete pipelines. Two popular and freely available architectures for natural language processing are the General Architecture for Text Engineering (GATE) (Cunningham *et al.* 2011) and the Unstructured Information Management Architecture (UIMA) (Ferrucci and Lally 2004). The contribution of this work, UIMA Ruta, is integrated in UIMA, as the name already indicates. Since UIMA Ruta makes extensive use of its main concepts, they are shortly introduced in the following paragraph.

[2] http://uima.apache.org/ruta.html

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

UIMA (Ferrucci and Lally 2004) is a flexible and extensible framework for the analysis and processing of unstructured data and text in particular. It directs its attention especially to the interoperability of components and the scale-out functionality for vast amounts of data. The components of the framework are called analysis engines and are specified in a descriptor that provides information about their implementation, configuration, and capabilities. The analysis engines communicate in a pipeline by adding or modifying the meta information stored in the Common Analysis System (CAS), which contains the currently processed document. This information is represented by typed feature structures and is ordered in indexes for an efficient access. The type of the feature structure defines its semantic and its additional features, which consist of primitive values or other feature structures. The available set of types and their inheritance is specified in type system descriptors. The most common type of a feature structure is the annotation that defines two additional features, begin and end, assigning its type and additional features to a span of text. Most analysis engines create new annotations or modify existing ones in order to represent the result of their analysis.

UIMA is only a framework and does not ship a rich selection of components. There are, however, component repositories like DKPro (Gurevych *et al.* 2007) that provide analysis engines of well-known components for natural language processing. Two prominent applications that built upon UIMA are the DeepQA system Watson (Ferrucci *et al.* 2010) and the clinical Text Analysis and Knowledge Extraction System (cTAKES) (Savova *et al.* 2010).

2.2 *Rule languages*

The Common Pattern Specification Language (CPSL) (Appelt and Onyshkevych 1998) is the result of the effort of different researchers for defining a system-independent language for information extraction. It defines a common formalism for the representation of finite-state grammars. The Java Annotation Patterns Engine (JAPE) (Cunningham *et al.* 2000) provides FST over annotations based on regular expressions. It is probably the most noted implementation of the CPSL specification with a few differences and extensions. JAPE is part of the GATE ecosystem, which led to its wide-spread use. Following the CPSL specification, a JAPE grammar consists of a set of phases, which are sequentially executed. A phase itself is composed of a set of rules that are compiled into one FST. The actual matching strategy of the rules can be specified at the beginning of the phase by the control style, which uses priorities to determine applicable rules and selects the next position after a match. The spans of newly created annotations are determined by labels in the condition part. The action part allows the inclusion of Java code in order to avoid limitations of the CPSL specification.

SProUT (Drozdzynski *et al.* 2004) (shallow processing with unification and typed feature structures) combines the ideas of FSTs, typed feature structures and unification-based grammars. The rules defined in their language XTDL are regular expressions over typed feature structures and allow to define coreference constraints

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

between the different elements of the rule. The consequences of the rules can apply externally defined operators. For execution, the rules are transformed into FSTs.

Boguraev and Neff (2010) presented an annotation-based finite-state transducer (AFST), which is able to navigate in dense annotation lattices. It extends the horizontal sequential patterns with navigation in the vertical direction, which enables the user to specify more constraints for the relationship of overlapping annotations. The system is completely based on UIMA and thus straightforwardly supports patterns over typed feature structures. AFST interprets the matching process of the FST as a linear path through the annotation lattice, which is based on the indexes and iterators of UIMA. If several annotations start at the same offsets, the annotation with the highest priority is selected. The exact matching behavior is specified by grammar-wide declarations. The spans of newly created annotations are determined with surrounding tags in the condition part. The AFST language also provides a small set of additional predicates.

SystemT (Chiticariu *et al.* 2010) is a good example for current trends in research about rule languages for information extraction. Its rule language AQL follows a more declarative approach regarding the definition of patterns and provides a syntax similar to SQL. The rules are not transformed into a FST but into an operator graph, which allows the selection of an optimized execution plan. Breaking up the strict left-to-right evaluation of the patterns, SystemT is able to achieve a much higher runtime performance. The resulting annotators can be integrated into UIMA. An investigation of the formal model underlying AQL can be found in Fagin *et al.* (2013).

Many other rule languages have been published in the recent years. CAFETIERE (Black *et al.* 2005) combines the strict sequential execution of regular expressions over annotations with basic coreference constraints. Its data model seems to allow only a disjunct partitioning of the document. HIEL (IJntema *et al.* 2012) is built upon JAPE and focuses on a compact rule representation for elements of an ontology. Xlog (Shen *et al.* 2007) is a rule language based on Datalog with embedded extraction predicates. It supports query optimization techniques for an improved runtime performance. The declarative language in PSOX (Bohannon *et al.* 2009) is based on an SQL-like syntax. The system focuses on the extensibility of its operator model, the explainability, and a scoring model for social feedback.

2.3 Development support

Many of the rule-based systems described in the last section also provide some development tools, which range from only basic syntax validation to editor support or testing of rules. The most notable tool is probably the development environment of SystemT (Chiticariu *et al.* 2011). It provides an editor with syntax highlighting and hyperlink navigation, an annotation provenance viewer, a contextual clue discoverer, a regular expression learner, and a rule refiner. In WizIE (Li *et al.* 2012), a process model is introduced that guides the developer in the different steps and enables novice developers to create high-quality applications (Yang *et al.* 2013).

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

The IBM Content Analytics Studio[3] is a UIMA-based development environment for the specification of rules in a drag and drop paradigm. ARDAKE[4] provides an environment for the integration of business and semantic rules in UIMA-based annotators. RAD (Khaitan *et al.* 2008) is a tool for Rapid Annotator Development. Its rules are based on inverted index operations (Ramakrishnan, Balakrishnan and Joshi 2006), which allows for a quick feedback of rule modifications.

3 UIMA Ruta language

The UIMA Ruta language is primarily a rule-based language for specifying patterns over annotations and additional consequences if the pattern successfully matches on a text position. The rules are applied sequentially in the order specified by the knowledge engineer. The language was incrementally extended with elements uncommon to rule languages such as control structures and variables. These two characteristics led to a perspective from which the UIMA Ruta language can be interpreted as a scripting language.

This section provides an introduction to the UIMA Ruta language and its rule matching process. The different aspects of the language and engineering approaches are illustrated by examples. Additional information about the exact syntax, provided language elements and further introductory examples can be found in the documentation of the system[5].

From a UIMA perspective, UIMA Ruta provides a generic analysis engine, which interprets the rule-based script files. Before the rules are applied, a tokenizer adds annotations to the document representing different classes of tokens such as capitalized words (CW, e.g., ‘Peter’), numbers (NUM), different kinds of punctuation marks (COMMA, PERIOD, . . .), and many more. These annotations can be utilized to define some initial rules, which, for their part, create new annotations that will be used by other rules. The types are part of a type hierarchy enabling the knowledge engineer to refer to lexical properties on different levels of abstraction.

3.1 Syntax and semantics

The following sections first describe the actual executable resource, the script file, before the syntax of rules in the UIMA Ruta language is defined. Along with the grammars, examples for rules and scripts are given in order to illustrate valid excerpts of the UIMA Ruta language.

3.1.1 Script definition

The general syntax of a UIMA Ruta script file is given in Grammar 1, which states that a script consists of an optional specification of the package followed by an

[3] formerly named IBM LanguageWare Resource Workbench: http://www.alphaworks.ibm.com/tech/lrw
[4] http://www.ardake.com/
[5] http://uima.apache.org/d/ruta-current/tools.ruta.book.html

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

optional list of import declarations and an optional list of statements. The language supports four different kinds of imports: The keyword ‘TYPESYSTEM’ indicates the import of a UIMA type system descriptor whereby its defined types are made available in the script file. The keyword ‘SCRIPT’ includes an additional script and its known types for further usage. The remaining two keywords ‘ENGINE’ and ‘UIMAFIT’ import analysis engines, which can then be executed from within the script file. Following the imports, a list of statements constitutes the major part of the script file and its actual functionality. Three different groups of statements can be distinguished. Declarations define new UIMA types, new variables or external dictionaries. The definition of new types serves only for rapid development in the UIMA Ruta Workbench since it avoids switching to other tools. The other two declarations define new elements of the UIMA Ruta language itself. The block statement is a script-like control structure that provides special functionality for the knowledge engineer such as procedures, definition by cases or restriction of the window the rules are applied in. This construct and similar language elements are described in Section 3.4. The remaining kind of statement, the rules, provides the actual functionality of the script file and is described in the next section after an example of a common composition of a script file.

Grammar 1. Simplified grammar of the script syntax.

```
⟨script⟩          ::= ⟨package⟩? ⟨import⟩* ⟨statement⟩*

⟨import⟩          ::= (’TYPESYSTEM’ | ’SCRIPT’ | ’ENGINE’ | ’UIMAFIT’)
                      ⟨identifier⟩ ’;’

⟨statement⟩       ::= ⟨declaration⟩ | ⟨rule⟩ | ⟨block⟩

⟨declaration⟩     ::= ⟨TypeDeclaration⟩
                    | ⟨VariableDeclaration⟩
                    | ⟨DictionaryDeclaration⟩

⟨block⟩           ::= ’BLOCK’ ’(’ ⟨identifier⟩ ’)’ ⟨ruleElement⟩ ’{’ ⟨statement⟩+ ’}’
```

Example 1 contains a script with diverse language elements for parsing bibliographic references. The example starts with a specification of the package of the script file, which is applied for defining the namespace of newly defined types and blocks. In lines 3 and 17, two comments provide an explanation of the corresponding environment. From lines 4 to 8, different global imports are added starting with importing a type system descriptor making its type definitions available. Then, three additional scripts are included and finally an external analysis engine is imported in line 8. Line 10 contains a type declaration. The following lines 11 to 15 provide simple rules, which execute the imported analysis engine and script files on the current document (e.g., Year as abbreviation of uima.ruta.example.Year). From line 18 onwards, a block declaration applies a rule for each annotation of the type Reference. This example illustrates the usage of the UIMA Ruta rule language for the definition of a pipeline with different components and some additional post-processing. Solving different engineering tasks like the combination of components or the declaration of new types is an important aspect for rapid development and avoids switching to other tools.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

```

PACKAGE uima.ruta.example;

// import the types of this type system
TYPESYSTEM types.BibtexTypeSystem;
SCRIPT uima.ruta.example.Author;
SCRIPT uima.ruta.example.Title;
SCRIPT uima.ruta.example.Year;
ENGINE uima.ruta.example.SegmentationEngine;

DECLARE Reference;
Document{-> CALL(SegmentationEngine)};

Document{-> CALL(Year)};
Document{-> CALL(Author)};
Document{-> CALL(Title)};

// create bibtex annotation
BLOCK(forEach) Reference{} {
        Document{-> CREATE(Bibtex, "author" = Author,
                "title" = Title, "year" = Year)};
}
```

Example 1. UIMA Ruta script pipeline for parsing bibliographic references.

3.1.2 Rule definition

The syntax of rules in the UIMA Ruta language is specified in Grammar 2. A rule commonly consists of a sequence of rule elements followed by a semicolon indicating the end of the rule. Simple regular expression rules and conjunctions of rules are two special kinds of rules. A rule element itself consists of at least a mandatory match reference, which creates a connection to the document by matching on a text fragment. This connection can be specified by five different kinds of references. A type expression represents a UIMA type and is the most common kind of match reference. The rule element matches on the annotation of the given type and, therefore, on the position covered by the annotation. The feature expression is an extension of the type expression by adding additional constraints on the feature values of the matched annotation or by referring to the annotations stored as feature values. A string expression represents a character sequence and enables the rule element to match directly on the text passage of the document with the identical character sequence. The match reference of a rule element can also consist of composed rule elements for a sequential, conjunctive or disjunctive grouping. The last kind of match reference is the wildcard, which provides a placeholder for any kind of text passage.

A rule element can be extended with several optional language elements. The anchor marker ‘@’ in front of a rule element will be discussed in Section 3.2. The optional quantifier part specifies how often the rule element may match or has to match for a successful rule match. It supports the four most common quantifiers known by regular expressions (‘?’, ‘*’, ‘+’, min/max), each in a greedy and reluctant form. A rule element without a quantifier has to match exactly once.

After the optional quantifier, curly brackets contain lists of optional conditions and actions, which are separated by an arrow ‘->’. Conditions are binary predicates and specify additional constraints on the matched position that need to be fulfilled

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

Grammar 2. Simplified grammar of the rule syntax.

```
⟨rule⟩              ::=  ( ⟨ruleElement⟩+ | ⟨regExpRule⟩ | ⟨conjunctRules⟩ ) ';'

⟨ruleElement⟩       ::=  '@'? ⟨matchReference⟩ ⟨quantifier⟩? ( '{' ⟨conditions⟩?
                         '->' ⟨actions⟩? '}' )? ⟨inlinedRules⟩?

⟨matchReference⟩    ::=  ⟨typeExpression⟩ | ⟨stringExpression⟩
                     |   ⟨featureExpression⟩ | ⟨composedRE⟩ | ⟨wildCard⟩

⟨composedRE⟩        ::=  '(' ⟨ruleElement⟩+ ')'
                     |   '(' ⟨ruleElement⟩ ( '&' ⟨ruleElement⟩ )+ ')'
                     |   '(' ⟨ruleElement⟩ ( '|' ⟨ruleElement⟩ )+ ')'

⟨quantifier⟩        ::=  '?' | '??' | '*' | '*?' | '+' | '+?'
                     |   '[' ⟨numberExpression⟩ ',' ⟨numberExpression⟩ ']'
                     |   '[' ⟨numberExpression⟩ ',' ⟨numberExpression⟩ ']' '?'

⟨conditions⟩        ::=  ⟨condition⟩ ( ',' ⟨condition⟩ )*

⟨actions⟩           ::=  ⟨action⟩ ( ',' ⟨action⟩ )*

⟨condition⟩         ::=  ConditionKeyword '(' ⟨expression⟩ ( ',' ⟨expression⟩ )* ')'

⟨action⟩            ::=  ActionKeyword '(' ⟨expression⟩ ( ',' ⟨expression⟩ )* ')'

⟨inlinedRules⟩      ::=  ( '->' | '<-' ) '{' ⟨rule⟩ + '}'
```

for a successful match of the rule element. Actions represent the consequences of the rule and are only applied if the complete rule matched successfully. Conditions and actions both start with a keyword indicating its type followed by a list of expressions in parentheses determining possible arguments and configurations. The last optional part of a rule element, the inlined rules, is discussed in Section 3.4. Examples of rules in different notations are given in the following. They are semantically identical, but represent the preferences of different rule engineers.

```
 1  // three notations of the same rule that matches on
 2  // texts like ``Dec. 2004'', ``July 85'' or ``11.2008''
 3  ANY{INLIST(MonthsList) -> MARK(Month), MARK(Date,1,3)}
 4      PERIOD? NUM{REGEXP(".{2,4}") -> MARK(Year)};
 5
 6  (ANY{INLIST(MonthsList) -> Month} PERIOD?
 7      NUM{REGEXP(".{2,4}") -> Year}){-> Date};
 8
 9  ANY{INLIST(MonthsList)} PERIOD? NUM{REGEXP(".{2,4}")
10      -> MARK(Month,1), MARK(Year,3), MARK(Date,1,3)};
```

Example 2. Three different notations of the same rule for detecting dates: old fashioned (line 3+4), compact (line 6+7), and traditional (line 9+10). Years with three digits are allowed.

The first rule in Example 2 (lines 3 and 4) consists of three rule elements. The first one (ANY. . .) matches on every token, which covers text that occurs in the word list MonthsList. The second rule element (PERIOD?) is optional and does not need to be fulfilled, which is indicated by its quantifier '?'. The last rule element (NUM. . .) matches on numbers that fulfill the regular expression REGEXP('.{2,4}') and have a length of at least two characters to a maximum of four characters. If this rule successfully matches on a text passage, then its three actions are executed: An annotation of the type Month is created for the first rule element, an annotation

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

Conditions	AFTER, AND, BEFORE, CONTAINS, CONTEXTCOUNT, COUNT, CURRENTCOUNT, ENDSWITH, FEATURE, IF, INLIST, IS, LAST, MOFN, NEAR, NOT, OR, PARSE, PARTOF, PARTOFNEQ, POSITION, REGEXP, SCORE, SIZE, STARTSWITH, TOTALCOUNT, VOTE
Actions	ADD, ADDFILTERTYPE, ADDRETAINTYPE, ASSIGN, CALL, CLEAR, COLOR, CONFIGURE, CREATE, DEL, DYNAMICANCHORING, EXEC, FILL, FILTERTYPE, GATHER, GET, GETFEATURE, GETLIST, GREEDYANCHORING, LOG, MARK, MARKFAST, MARKFIRST, MARKLAST, MARKONCE, MARKSCORE, MARKTABLE, MATCHEDTEXT, MERGE, REMOVE, REMOVEDUPLICATE, REMOVEFILTERTYPE, REMOVERETAINTYPE, REPLACE, RETAINTYPE, SETFEATURE, SHIFT, TRANSFER, TRIE, TRIM, UNMARK, UNMARKALL

Fig. 1. (Colour online) List of conditions and actions currently available in UIMA Ruta.

of the type Year is created for the last rule element, and an annotation of the type Date is created for the span of all three rule elements. If the word list contains the correct entries, then this rule matches on strings like ‘Dec. 2004’, ‘July 85’ or ‘Nov. 2008’ and creates the corresponding annotations.

The language also supports syntactic sugar that allows one to specify conditions and actions using expressions without keywords. The knowledge engineer can use boolean expressions, such as boolean variables, or feature-match expressions in order to formulate compact conditions. As for actions, a type expression is able to replace a MARK action and feature-assignment expressions are able to modify the values of matched feature structures. The second rule in Example 2 (lines 6 and 7) provides an alternative of how the first rule can be rewritten without MARK actions.

Usually, the conditions and actions are clearly separated in well-known rule languages. While the conditions make up the left-hand part, the actions follow in a right-hand part after a distinctive separator. In the UIMA Ruta language, actions can occur at each rule element after the list of conditions and are not located solely at the end of the rule as it would be expected. Allowing actions to be attached to the rule element, which supplies the context for its consequences, provides various advantages and results in a more compact rule representation. However, the actions can be detached and listed at the last rule element in most situations, simulating the traditional composition. The third rule in Example 2 (lines 9 and 10) provides an example of how the first rule can be rewritten with the actions located at the end of the rule.

The conditions and actions in the previous examples reflect only a small sample of the available constructs. The language provides more than 25 different conditions and more than 40 different actions, which enables the knowledge engineer to tackle different annotation tasks efficiently. An overview of the available actions and conditions is given in Figure 1. Actions, for example, cannot only add new annotations, but are also able to remove (UNMARK) or modify existing ones (SHIFT), or execute other components (EXEC). Special actions are also applied for creating relations between entities or simply copying feature values as shown in Example 3.

```
Token{-> CREATE(Container, "pos" = Token.posTag, "lemma" = Lemma.value,
  "token" = Token)};
```

Example 3. A simple rule for copying feature values and assigning annotations to features. A new annotation of the type Container is created, which stores different information of the underlying annotations as feature values.

The rule creates a new annotation of the type Container for each Token annotation. Furthermore, three features are automatically filled with values: The value of the feature ‘posTag’ of the Token annotation is assigned to the feature ‘pos’ of the Container annotation. The value of the feature ‘value’ of a Lemma annotation with the same offsets as the matched Token annotation is assigned to the feature ‘lemma’ of the Container annotation. Finally, the complete Token annotation is stored in the feature ‘token’.

Different users have different use cases and require specialized language constructs for efficiently implementing their rule-based applications. UIMA Ruta provides a plugin concept for extending the language by additional actions, conditions, functions, and even blocks, which modify rule execution. The knowledge engineer is not restricted to the available list of language elements, but can adapt and optimize the language for her use cases.

3.2 Inference

The description of the syntax and semantics in the last section provided an overview on the specification of valid rules and their meaning. This section now considers how the rules are applied. Before the matching algorithm is introduced, the order of rule application is discussed.

3.2.1 Rule execution

UIMA Ruta rules are applied in an imperative manner, one rule after another, in the same order they occur in the script file. This leads to an interpretation of the language as cascaded FSTs, whereas each transducer corresponds to one rule. Other rule-based languages for information extraction compile the rules of one phase into one FST in order to avoid unnecessary and duplicate inference steps. Although sequential execution provides some disadvantages, the advantages prevail in the focus of the system, which is rapid development of rule-based information extraction applications.

The disadvantages consist mainly in the possibility of a decreased performance for large rule sets and in the missing truth maintenance between rules. However, truth maintenance is hardly supported by automata-based languages in general. The performance issue compared to rules compiled in one single FST occurs especially if many rules start with the same match reference resulting in a variety of identical and redundant operations. If, however, the rules have no joint match references, the performance differs only marginally since the rules share no states in the automaton. The absence of truth maintenance potentially leads to an increased number of rules. If a rule activates or negates the preconditions of previous rules, then their postconditions are not automatically executed or revoked. The knowledge engineer has to add additional rules for propagating the desired effect like in other languages.

The advantages of this kind of imperative rule execution can particularly be found in its simplicity, which is important for rapid development of rule sets and can be essential for inexperienced users or users not familiar with rule-based systems. The user does not have to consider side effects to previous rules and snares of dependent rules. Due to the missing truth maintenance, the rules of a script have

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

to be structured correspondingly to their dependencies, which results in a clear and comprehensive set of rules. The absence of truth maintenance also implies an improved performance because the match references and conditions of other rules do not need to be reevaluated after a rule added or removed annotations. Furthermore, the ability to revoke the postconditions of rules either confines the rule language or increases its complexity. The usage of rule scripts with truth maintenance as a prototyping language for the definition of pipelines, for example, is only possible if the operations of arbitrary components can be taken back. The linear execution of rules makes the definition of pipelines possible in the first place. Another advantage consists in an easier explanation of the rule inference, which enables the user to identify the causes of undesired rule behavior quickly.

3.2.2 Rule matching

Rules in UIMA Ruta are atomic statements concerning the inference, as pointed out in the last section. A rule itself can be interpreted as a FST. For the description of the rule matching in UIMA Ruta a pseudo code algorithm is utilized.

Algorithm 1 Pseudo-code of the rule matching algorithm in UIMA Ruta.

```
procedure STARTMATCH
    rule element $e$ ← identify starting rule element
    CONTINUEMATCH($e$)
end procedure

procedure CONTINUEMATCH(rule element $e$, optional position $p$, optional rule match $o$)
    if position $p$ is given then
        anchors $a$ ← all valid positions next to position $p$
    else
        anchors $a$ ← all valid positions for the rule element $e$
    end if
    for all positions $i$ of anchors $a$ do
        rule match $m$ ← new alternative of rule match $o$ or new rule match
        validate match reference and conditions of $e$ on position $i$ for rule match $m$
        rule element $n$ ← NEXTELEMENT(rule element $e$, rule match $m$)
        CONTINUEMATCH(rule element $n$, position $i$, rule match $m$)
    end for
end procedure

function NEXTELEMENT(rule element $e$, rule match $m$)
    if quantifier of $e$ indicates further repetition then
        return rule element $e$
    else
        rule element $n$ ← identify rule element next to $e$
        if rule element $n$ exists and rule match $m$ is valid then
            return rule element $n$
        else
            DONEMATCHING(rule match $m$)
        end if
    end if
end function

procedure DONEMATCHING(rule match $m$)
    if rule match $m$ is valid then
        apply all actions of rule
    end if
end procedure
```

The rule matching of UIMA Ruta is specified in Algorithm 1. The rule starts to match by calling the procedure STARTMATCH. First of all, the rule element

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

is determined, which initiates the rule matching process. Normally, this is the first rule element of the rule resulting in a left-to-right match. If the match reference of the selected rule element consists of a composed rule element, then the starting rule element is determined in the list of contained elements. The beginning rule element is requested to continue the matching process with the procedure continueMatch.

The procedure continueMatch provides one to three arguments: the current rule element, an optional position and an optional rule match. If this method is called by the procedure startMatch, then only one argument is given. First of all, valid positions for the rule element are determined. Either all valid positions, e.g., annotations of the type of the match reference, are listed, or only valid positions next to the provided position in the arguments are listed. The strategy for selecting the next position can be configured. For each possible position of this list, several operations are performed. First, a new rule match is created, which stores the current state of the matching process, e.g., already evaluated positions. If a rule match was provided by the arguments of the procedure, then a copy is created, representing a new alternative rule match. The position is validated concerning the match reference, the conditions and the additional rules inlined as conditions. UIMA Ruta is, therefore, able to handle positions where multiple annotations begin. If the rule element matched successfully, then the next rule element is identified by using the function nextElement. The next rule element is then requested to continue with the matching process at the new position.

The function nextElement not only provides the next rule element, but also terminates the matching process if no remaining rule elements can be found. The function initially checks whether the quantifier of the rule element allows repetitions, and returns the current one as appropriate. Reluctant and optional quantifiers are neglected in the pseudo code for simplicity. If the rule element has matched sufficiently, then the next rule element is determined. This is usually the rule element following the current one. The last procedure doneMatching simply validates if the current rule match was successful and then applies all actions and additional rules inlined as actions on the positions stored in the rule match. The matching algorithm tracks a rule match until it terminates and possibly applies the actions before an alternative match is considered. This facilitates the specification of useful rules.

The description of the matching algorithm mentioned that the starting rule element is normally the first rule element of the rule, which can decrease the performance of the matching process for certain rules to some extent. This problem is illustrated with two rules in Example 4 that match on the second last token.

```
ANY LastToken;
ANY @LastToken;
```

Example 4. Two simple rules that match on a token followed by a LastToken annotation. While the first rule has to investigate every token, the second rule starts to match with the second rule element and requires less index operations.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

The first rule investigates each token of the document starting with the first one, which results in many unnecessary rule matches. The second rule provides a start anchor, indicated by the symbol '@'. This optional symbol enables the user to manually specify the starting rule element. The rule begins with the annotation of the type LastToken and continues the match with the previous rule element, which results in a right-to-left matching and in only one rule match. The UIMA Ruta language also provides an option called 'dynamic anchoring' that automatically determines the starting element using a heuristic applied on the amount of involved annotations. The option can be activated in the configuration parameters or by other rules in the script. The possible occurrences of matching references of the rule elements are compared and the rule element with the least amount of initial matches is selected. The heuristic also includes the quantifier and composed rule elements. Furthermore, a penalty for the reverse matching direction can be specified. Dynamic anchoring is a newer feature and thus not yet activated by default.

As a special kind of match reference, the wildcard '#' can be used to optimize the rule matching performance further. The two rules in Example 5 create an annotation of the type Sentence for all text passages that are surrounded by periods. While the first rule matches on one token (ANY) after another until the next occurrence of a period, the second rule matches directly on the next period found in the UIMA index. This reduces the amount of considered positions and also provides a compact representation.

```
PERIOD ANY+?{-> Sentence} PERIOD;
PERIOD #{-> Sentence} PERIOD;
```

Example 5. Two equivalent rules for annotating text between two periods. While the first rule needs to match on each token (ANY), the second rule just searches for the next period resulting in less UIMA index operations.

The general performance concerning execution time of the UIMA Ruta rule-based script mainly depends on the amount of index operations in the UIMA framework, and, related to this, on the amount of rule matches and the involved conditions. Similar to programming languages, it is possible to implement slow and fast rule sets for solving the same problem. Since the language was especially designed to support rapid development, the user normally does not stress performance issues in the first place. If, however, the execution time of a rule script needs improvement, many possibilities exist.

This section introduced a few options to reduce the amount of rule matches or index operations. The user can also make use of efficient dictionaries or can profile the rule execution in order to identify bottlenecks (cf. 4.2).

3.2.3 Beyond sequential matching

The matching process described so far only considers rule elements as sequential patterns. The language also supports rule elements that need to match on the same position. As introduced in Section 3.1.2, composed rule elements are not only able to specify sequential patterns, but also disjunctive and conjunctive rule elements. The list of disjunctive rule elements separated by the symbol '|' specify that at least one of the rule elements needs to match in order to obtain a successful match of the composed rule element. Analogously, all of the conjunctive rule elements separated by the symbol '&' need to match in order to continue the matching process. These language elements can be applied for verifying different aspects on the same position that cannot be represented by a combination of conditions.

Another language construct that does not represent a strict sequential pattern is specified by the symbol '%', which is applied for connecting two rules. The resulting rule builds a conjunction of both rules and its actions are only applied if both rules have successfully matched. While conjunctive rule elements ('&') have to match on the same position, the connected rules may match independently of each other on different positions in the current window or document.

The rule in Example 6 consists of two connected rules and matches only if there is an arbitrary capitalized word followed by a number in the document and if there is also some arbitrary lower-cased word followed by a number.

```
CW NUM % SW NUM;
```

Example 6. A conjunction of two simple rules. The complete rule matches only if both rules are able to match independently of each other.

3.3 Visibility and filtering

The UIMA Ruta language and its inference are designed to provide an instrument for solving different text processing and information extraction tasks. One step in this direction is the possibility of defining patterns not only over token but over arbitrary types of annotations. Rules can, therefore, be applied for matching sequences of tokens, but also sequences of paragraphs. Another feature provided by UIMA Ruta is the specification of the visibility that determines which kinds of text passages represented by annotations are accessible by the rules. While one type of text is important in one use case, it should be ignored in other applications. If the rules are applied, for example, to label sequences of tokens, then the whitespaces between tokens are of minor interest and the user should be able to ignore them in the definition of the rule set. However, if the rules are built in order to parse identifiers or indentation of tables, whitespaces are essential and need to be included in the pattern. Another example is the processing of headlines in a document. In a sequential pattern over headline annotations, the paragraphs do not need to be considered in the specifications of rules.

Rule-based languages often specify the types available in a phase whereas annotations of other types are automatically skipped. The UIMA Ruta language provides a more complex and dynamic concept of visibility. Here, text positions covered by annotations of specific types are invisible. This leads to a coverage-based visibility concept instead of a type-based one. All annotations and their covered text passages that start or end with a type specified in the set of filtered types are not accessible by the rules. This means that these invisible positions will be skipped when the next position for the rule match is determined. The set is calculated using three lists. The *default* list is specified in the configuration parameters and generally contains types for whitespace and markup. The elements of the *filtered* list are added to this list. Afterwards, the elements of the *retained* list are removed. The *filtered* and *retained* lists can be modified by actions so that the knowledge engineer is able to adapt the rule inference to her current requirements directly in a UIMA Ruta script. The exact behavior of rules facing invisible annotations is explained with Examples 7 and 8.

```
1 W NUM;
2 Document{-> RETAINTYPE(SPACE,MARKUP)};
3 W NUM;
```

Example 7. Two identical rules that match on different text positions due to the changed filtering settings in the second rule. The second rule is sensible to markup and whitespaces in its sequential constraint.

The first rule in Example 7 matches on text fragments like ‘Dec<br>2004’, ‘May1999’, or ‘July 85’ since whitespaces and markup are filtered by default. The second rule (line 2) changes the filtering settings by adding the types for space and markup annotations to the *retained* list, which makes them visible again. The last rule is identical to the first one, but matches only on the text fragment ‘May1999’ since the matching process was unable to find a valid subsequent position.

```
1 Sentence;
2 Document{-> RETAINTYPE(MARKUP)};
3 Sentence;
4 Document{-> FILTERTYPE(Headline)};
5 Sentence;
6 Document{-> RETAINTYPE, FILTERTYPE};
```

Example 8. Three rules for matching on sentences. The other rules change the filtering setting resulting in different matches on sentences.

Example 8 contains three identical rules that match on a sentence and three rules that change the filtering settings. The first rule matches on sentences that do not start with a whitespace or markup annotation due to the default filtering settings. The second rule enables matching on markups. The third rule (line 3) also matches on sentences that start with a markup element. The fourth rule hides headlines and, thus, the fifth rule is not able to match on sentences that start with or are part of a headline. The last rule resets the filtering settings to its default values.

3.4 Blocks and inlined rRules

The implementation of annotation tasks using only a plain rule language often leads to rule sets that are hard to interpret by a human. This is caused by the lack of control structures, which are compensated by additional conditions or activation rules. Control structures can be very useful in rule-based information extraction applications. Examples for those elements are the restriction of the rule match to a certain window, further modularization, conditioned execution of rule sets or application of rules for each occurrence of an annotation.

The UIMA Ruta language provides the BLOCK construct for these use cases. The syntax of blocks was already specified in Section 3.1.1. A block construct starts with the keyword ‘BLOCK’ followed by an identifier that is utilized in case the block is invoked by another rule. The main part, the head, is a rule element, which specifies the functionality of the block. The body of the construct finally contains a list of statements, e.g., rules. The rules within the block are only applied in the context of the matches of the rule element in the head. If the rule element did not match, then the contained rules are not applied at all, which corresponds to a conditioned statement. If the rule element matches on several annotations, then the contained rules are applied once for each matched annotation and only within this annotation, which corresponds to an iteration over the annotations and a restriction of the context. These use cases are illustrated with examples.

```
BLOCK(German) Document{FEATURE("language", "de")} {
    // rules for german documents
}
```

Example 9. A conditioned statement using the block construct. The contained rules are only applied if the language of the document is set to ‘de’.

Example 9 provides a block construct that applies the contained rules only if the language of the document was set to ‘de’.

```
BLOCK(ForEach) Sentence{} {
    // ... do something
}
```

Example 10. Iteration over annotations of the type Sentence. The contained rules are applied for each sentence and only in the window of the current sentence.

The block in Example 10 applies the contained rules on each sentence. Rules that try to match over the boundaries of a sentence will automatically fail. Within the body of the block, the type Document refers to the current sentence rather than to the whole document.

Another language element that provides similar functionality is an extension of a rule element, the so-called inlined rules (cf. Section 3.1.2). These occur in two manifestations, either interpreted as consequences indicated by the symbol ‘->’ or as preconditions (‘<-’). The former kind provides similar functionality as the block element, but can directly be utilized in more complex rules. If the rule matched successfully, then the inlined rules are applied in the context of the match of the rule element. The latter provides functionality for expressing more complex, nested conditions. Here, the rule itself matches successfully in the first place if one of the contained rules was able to match. Both extensions enable the knowledge engineer to specify complex patterns in a compact representation. Their syntax and semantics are illustrated with two examples.

The rule in Example 11 matches on an annotation of the type Prefix followed by a sentence annotation. If this match was successful, then the rule in line 2 is applied in the context of the matched sentence annotation. It creates an annotation of the type SentNoLeadingNP with the offsets of the matched sentence, if the sentence does not start with an annotation of the type NP.

```
1 Prefix Sentence->{
2     Document{-STARTSWITH(NP) -> SentNoLeadingNP};
3 };
```

Example 11. An example of an inlined rule interpreted as a postcondition. An annotation is created for each sentence if additional requirements are fulfilled.

Example 12 contains a rule element extended with an inlined rule interpreted as a precondition. The rule tries to match on each annotation of the type Sentence, but only succeeds if this sentence contains an annotation of the type NP followed by another NP.

```
Sentence{-> SentenceWithNPNP}<-{
    NP NP;
};
```

Example 12. An example of a rule element with an inlined rule interpreted as a precondition. An annotation is created only if the sentence contained two subsequent noun phrases.

3.5 Engineering approaches

The support of different engineering approaches for solving an annotation task is an important characteristic of a generic rule-based information extraction system. The formalization of rules based on only one engineering perspective may lead to inconvenient representations. The UIMA Ruta language was in particular designed to provide a generic pattern formalism that enables the user to solve annotation tasks with different approaches. Among other things, it is achieved with special conditions and actions that encapsulate the necessary functionality. In the following, a selection of approaches are discussed and illustrated with examples. These approaches do not have to be applied separately, but can also be mixed at each stage.

3.5.1 Classical approaches

Many classical approaches for rule-based information extraction can be identified. The simplest one is called *candidate classification*, which generates possible candidate annotations and then assigns a specific type if the candidate holds certain properties. The rule in Example 13 considers each annotation of the type paragraph and labels it as a headline if it is mostly bold, underlined and ends with a colon.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

```
Paragraph{CONTAINS(Bold, 90, 100, true),
    CONTAINS(Underlined, 90, 100, true), ENDSWITH(COLON)
    -> MARK(Headline)};
```

Example 13. Candidate classification with UIMA Ruta rules. The rule classifies a paragraph as a headline if it is ninety to hundred percent covered by Bold and Underlined annotations, and ends with a colon

Annotations of interest can be approached in a *bottom-up* or a *top-down* manner. The former approach starts by identifying small parts of the targeted annotations and successively composes them until the desired annotation can be specified. The latter approach starts with more general annotations and refines or reduces the considered positions until the annotations of interest are found. Example 14 provides an example for labeling the author section of a scientific reference using a bottom-up approach. The first rule creates annotations for name initials, which consist only of one upper-case letter followed by a period. The second rule specifies names as a capitalized word followed by a comma and a list of name initials. The last rule finally combines a listing of names to the annotations of the type Author.

```
(CW{REGEXP(".")} PERIOD){-> Initial};
(CW COMMA Initial+){-> Name};
(Name (COMMA Name)*){-> Author};
```

Example 14. Bottom-up approach for labeling author sections. The first rule detects initials, the second rule identifies names, and the third rule combines names to authors.

Rules that consider the content of an annotation are sometimes hard to specify. However, the boundaries of the targeted annotation are possibly easier to identify without facing the variety of patterns occurring within that annotation. An example of this *boundary matching* approach is given in Example 15. The first rule identifies the start of the author part of a reference and the second rule detects possible ends of the author section. The third rule creates an annotation for spans that starts with an annotation of the type AuthorStart and ends with an annotation of the type AuthorEnd.

```
Reference{-> MARKFIRST(AuthorStart)};
COLON{-> AuthorEnd} CW;
(AuthorStart # AuthorEnd){-> Author};
```

Example 15. Boundary matching approach for labeling author sections. First rule detects the start position, the second rule identifies the end position, and the third rule combines both for the complete annotation.

3.5.2 Transformation-based rules

Transformation-based rules are applied on already existing annotations and try to correct specific errors or defects. The usage of transformations can greatly ease the definition of patterns and accelerate the development of rule sets. The inclusion of all possible exceptions or negative preconditions in the rule that creates the interesting annotations leads to confusing rule constructs. These exceptions can be neglected initially, if later transformation-based rules remove annotations that should have been excluded. The transformation-based approach can also be beneficial in many other scenarios. One example is the usage of transformations for domain adaptation or improving the accuracy of arbitrary models.

The UIMA Ruta language provides several actions for modifying existing annotations. The action UNMARK removes annotations of the given type, the action SHIFT changes the offsets of an annotation dependent on the other rule elements and the action TRIM reduces the span of an annotation. The first rule in Example 16 removes annotations of the type Headline, if they contain no words at all. A previous rule identified headlines using the layout of the document, but did not include a condition validating their contents. If only one rule was responsible for annotating headlines, the additional precondition is easily added. If the annotations of the type Headline are, however, created by twenty different rules, then the additional condition has to be added twenty times, which decreases the readability and aggravates possible refactorings of the script. The second rule in Example 16 expands the span of an annotation of the type Person if it is preceded by the word ‘Mr’ and an optional period.

```
Headline{-CONTAINS(W) -> UNMARK(Headline)};
"Mr" PERIOD? @Person{-> SHIFT(Person,1,3)};
```

Example 16. Two examples for transformation-based rules. The first rule deletes headlines without words and the second rule includes text like ‘Mr.’ in Person annotations.

3.5.3 Scoring rules

It is sometimes not possible to specify a combination of properties in one rule. In some situations the knowledge engineer wants to weight different aspects for dealing with uncertainty. The UIMA Ruta language provides a special action and condition for such use cases. The action MARKSCORE adds a heuristic score for a certain kind of annotation and the condition SCORE is able to evaluate this score for further processing. While scoring rules can help to solve problematic tasks in a compact manner, larger sets of scoring rules get increasingly hard to maintain. Example 17 contains a small example for the identification of headlines.

```
 1 STRING s;
 2 Paragraph{CONTAINS(W,1,5)->MARKSCORE(5,HeadlineInd)};
 3 Paragraph{CONTAINS(W,6,10)->MARKSCORE(2,HeadlineInd)};
 4 Paragraph{CONTAINS(Bold,80,100,true)->MARKSCORE(7,HeadlineInd)};
 5 Paragraph{CONTAINS(Bold,30,80,true)->MARKSCORE(3,HeadlineInd)};
 6 Paragraph{CONTAINS(CW,50,100,true)->MARKSCORE(7,HeadlineInd)};
 7 Paragraph{CONTAINS(W,0,0)->MARKSCORE(-50,HeadlineInd)};
 8 HeadlineInd{SCORE(10)->MARK(Headline)};
 9 HeadlineInd{SCORE(5,10)->MATCHEDTEXT(s),
10         LOG("Maybe a headline: " + s)};
```

Example 17. Scoring rules for weighting different aspects of headlines. The rules create an annotation of the type Headline for paragraphs like ‘Diagnoses:’ since the first (line 2) and fifth rule (line 6) increase the score resulting in an overall score of 12. The rule in line 8 evaluates the score and creates a new annotation.

The rules from lines 2 to 7 weight different aspects indicative of headlines and assign a score to the annotation HeadlineInd. The first two rules specify that a paragraph with fewer words is more likely to be a headline. The third and fourth rules investigate the layout and the fifth rule assigns higher scores to paragraphs with many capitalized words. The sixth rule reduces the score for paragraphs that contain no words at all. The remaining rules evaluate the score and either create a new annotation of the type Headline, if the score exceeded the threshold of 10, or emit a message for less certain positions using a string variable defined in line 1.

In summary, the UIMA Ruta language facilitates the quick specification of comprehensible extraction knowledge. It supports a compact representation while still providing a high level of expressiveness for solving diverse tasks.

4 UIMA Ruta workbench

The development of rule-based applications for information extraction is first of all an engineering task. It can be a difficult, tedious and time consuming task, which depends on human resources and their qualification. Qualified rule engineers are potentially an expensive resource or only available to a limited extent. The engineers should therefore be supported by tools that ease the access to the rule-based systems and that improve the engineering experience in general. While improvements of the usability and an extensive documentation weaken the learning curve, the development environment should provide features that decrease the time and labor enabling the rapid development of rule sets. A (nonexhaustive) list of those features for supporting the rule engineer can be identified:

Clear visualization of different language elements The visualization of language elements by highlighting different syntactic constructs provides a clear overview of the rule set. Semantic highlighting of the occurrences of some language elements further improves readability.

Direct feedback on defective rules Writing rules is an error-prone process. The development environment should notify the user instantly about syntax errors, typing errors and misplaced constructs. The engineering process is decelerated if feedback about erroneous rules is not given before the rules are applied.

Shortcuts for editing The development environment should provide functionality for recurring elements either by completion of the statements of the user or by templates for favored constructs. In general, the effort of writing rules should be minimized.

Adaptable to process model of user The rule engineering task can be approached with different process models. The user should be able to adapt and optimize the development environment to her own process model.

Easy introspection of results When the engineered rules are applied on exemplary documents, the resulting annotations should be visualized in an clear and accessible way. The user should be able to directly investigate different aspects of the annotated documents.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

Explanation of rule execution The application of rules on a document should not be opaque. The user should be able to comprehend and track every step of the rule inference in order to identify and correct undesired behavior.

Automatic validation of the rule set's correctness Manual verification of rules and validation of their results is tedious and time consuming and potentially has to be performed every time the rule set is extended or refined. This task should be automated by specifying test cases, and informative reports should be provided to the user.

Support for automatic induction of rules The development environment should support the user in the specification of rule sets. In case additional data is available, it can be utilized in order to propose rules that solve a specific annotation task.

The UIMA Ruta Workbench tries to provide a development environment and additional tooling, which cover all of these features. It is implemented as a rich client application extending the Eclipse platform[6], which allows the user to arrange the different features and tools according to her preferences. The available tooling of the Eclipse platform can be directly utilized in the UIMA Ruta Workbench, e.g., version control of script files, which allows collaborative development[7]. Figures 2–4 depict various tools in a compact layout. The large amount of features and additional tooling leads to an increased complexity of the system. Thus, the UIMA Ruta Workbench is not easily accessible for inexperienced users, but favors trained engineers, who are able to take advantage of all features for an increased productivity. The following sections highlight how a strong tooling support can render the rapid development of rule-based information extraction systems possible, and refer to the figures in order to illustrate the different tooling support.

4.1 Basic development support

A development environment for rule-based information extraction applications should provide at least some basic features in the opinion of the authors, like the option to create and modify rules, the execution of these rules on a set of documents and the visualization of the annotations created by the rules. The central parts of the Workbench are the workspace with UIMA Ruta projects, the full-featured editor, the launch capacity for executing script files and the visualization of annotated documents based on the CAS Editor[8], which can also be used to create new annotations, e.g., for gold standard documents.

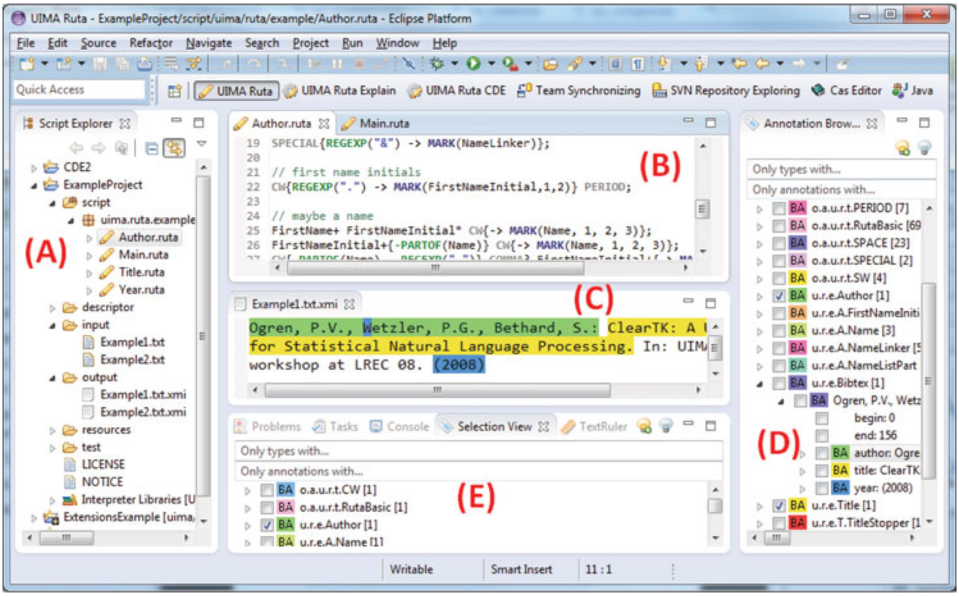
Figure 2 provides a screenshot of some views of the default perspective of the UIMA Ruta Workbench. Part (A) contains the list of UIMA Ruta projects in the workspace. A UIMA Ruta project applies a distinct folder layout: the script folder

[6] http://www.eclipse.org/

[7] If different rule engineers work on the same script files, then they require tools for sharing and investigating the performed changes. The UIMA Ruta Workbench is compatible with all prevalent SCM integrations like SVN or Git.

[8] http://uima.apache.org/d/uimaj-2.6.0/tools.html#ugr.tools.ce

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press



```
SPECIAL{REGEXP("&") -> MARK(NameLinker)};

// first name initials
CW{REGEXP(".") -> MARK(FirstNameInitial,1,2)} PERIOD;

// maybe a name
FirstName+ FirstNameInitial* CW{-> MARK(Name, 1, 2, 3)};
FirstNameInitial+{-PARTOF(Name)} CW{-> MARK(Name, 1, 2, 3)};
```

```
Ogren, P.V., Wetzler, P.G., Bethard, S.: ClearTK: A U
for Statistical Natural Language Processing. In: UIM
workshop at LREC 08. (2008)
```

Fig. 2. (Colour online) The UIMA Ruta Workbench: (A) *Script Explorer* with a UIMA Ruta project. (B) Full-featured editor for specifying rules. (C) *CAS Editor* for visualizing the results. (D) Overview of annotations sorted by type. (E) Annotations overlapping the selected position in the active CAS Editor.

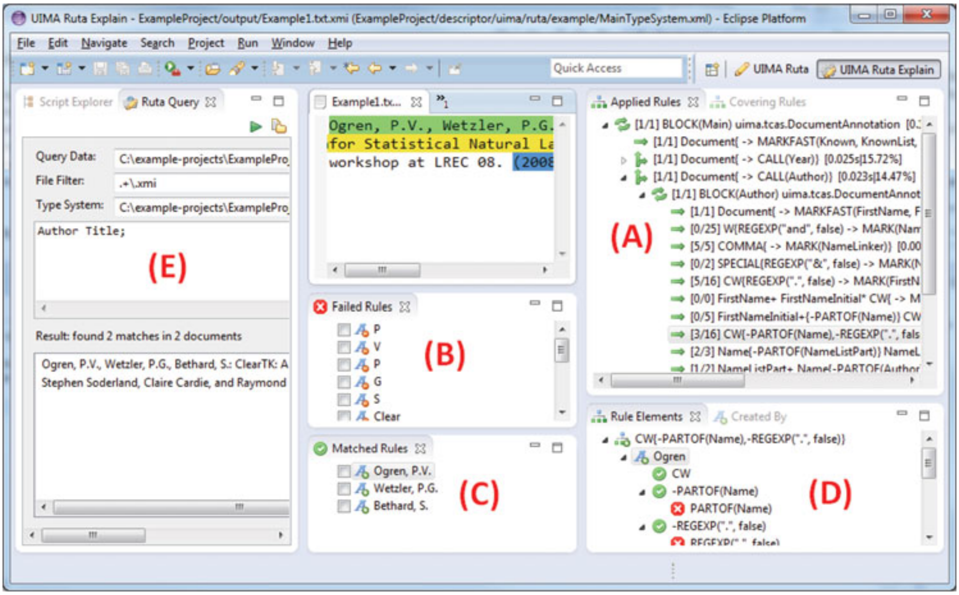
contains the rule-based script files, the descriptor folder the descriptor files of UIMA analysis engines and type systems, and the resources folder contains dictionaries and word lists. The Workbench automatically generates an analysis engine and type system descriptor for each script file. These descriptors can be utilized in arbitrary UIMA applications. When executing a script file, the documents in the input folder are processed by default and their results are stored in the output folder.

The user is able to add dependencies in a UIMA Ruta project pointing to other projects of the workspace. A dependency to a UIMA Ruta project enables the user to refer to its script files and leads to reusable subprojects. The Workbench takes care of the correct configuration of the generated analysis engines descriptors. A UIMA Ruta project can, however, also have dependencies to Java projects in the workspace. When a UIMA Ruta script is launched, then the classpath is automatically expanded by the dependencies, and analysis engines implemented in the same workspace can be utilized in a script file. This enables rapid prototyping across languages.

Part (B) in Figure 2 shows the full-featured editor. It provides editing support known by common programming development environments, especially syntax highlighting, semantic highlighting, syntax checking, auto-completion, template-based completion and more. Thereby, the user is optimally supported in writing or editing rules. Defective rules are instantly highlighted in the editor and the auto-completion proposes suitable language elements based on the current editing position.

For visualizing the results of the rule execution, the UIMA Ruta Workbench utilizes the CAS Editor (C) extended with additional views for an improved access

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press


UIMA Ruta Explain - ExampleProject/output/Example1.txt.xmi (ExampleProject/descriptor/uima/ruta/example/MainTypeSystem.xml) - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Quick Access
UIMA Ruta
UIMA Ruta Explain
Script Explorer
Ruta Query
Example1.tx...
Applied Rules
Covering Rules
Failed Rules
Matched Rules
Rule Elements
Created By
(A)
(B)
(C)
(D)
(E)

Query Data: C:\example-projects\ExamplePro

File Filter: .+\.xmi

Type System: C:\example-projects\ExamplePro

```
Author Title;
```

Result: found 2 matches in 2 documents

Ogren, P.V., Wetzler, P.G., Bethard, S.: ClearTK: A
Stephen Soderland, Claire Cardie, and Raymond

```
Ogren, P.V., Wetzler, P.G.
for Statistical Natural La
workshop at LREC 08. (2008
```

```
[1/1] BLOCK(Main) uima.tcas.DocumentAnnotation [0.:
    [1/1] Document{ -> MARKFAST(Known, KnownList,
    [1/1] Document{ -> CALL(Year)} [0.025s|15.72%]
    [1/1] Document{ -> CALL(Author)} [0.023s|14.47%]
        [1/1] BLOCK(Author) uima.tcas.DocumentAnnot
            [1/1] Document{ -> MARKFAST(FirstName, F
            [0/25] W{REGEXP("and", false) -> MARK(Nam
            [5/5] COMMA{ -> MARK(NameLinker)} [0.00
            [0/2] SPECIAL{REGEXP("&", false) -> MARK(N
            [5/16] CW{REGEXP(".", false) -> MARK(FirstN
            [0/0] FirstName+ FirstNameInitial* CW{ -> M
            [0/5] FirstNameInitial+{-PARTOF(Name)} CW
            [3/16] CW{-PARTOF(Name),-REGEXP(".", fals
            [2/3] Name{-PARTOF(NameListPart)} NameL
            [1/2] NameListPart+ Name{-PARTOF(Author
```

□ P
□ V
□ P
□ G
□ S
□ Clear

□ Ogren, P.V.
□ Wetzler, P.G.
□ Bethard, S.

```
CW{-PARTOF(Name),-REGEXP(".", false)}
    Ogren
        CW
        -PARTOF(Name)
            PARTOF(Name)
        -REGEXP(".", false)
            REGEXP(".", false)
```

Fig. 3. (Colour online) The UIMA Ruta Workbench: (A+B+C+D) A selection of views for the explanation of the rule execution: (A) The *Applied Rules* view for detailed report with profiling information. (B+C) Successful and failed matches. (D) The *Rule Elements* view displaying matches of rule elements and results of conditions. (E) The *Ruta Query* view for introspection in a collection of documents.

to the annotations. The *Annotation Browser* view (D) lists the annotations of a document sorted by their types. The *Selection* view (E) lists all annotations overlapping the position that is currently selected in the CAS Editor. Both views provide additional filtering options and enable the user to obtain a fast overview of the results of the rule execution and to investigate overlapping annotations at specific positions.

4.2 Explanation of rule execution

The explanation of rule execution is an essential feature for rule-based systems in general. Writing new rules is laborious, especially if the newly written rules do not behave as intended. The user has to be able to comprehend and track the behavior of the rules especially in order to identify undesired rule behavior. Answers to different questions need to be provided to the user: Did the rule match at all? Where did the rule match successfully or where did it failed to match? Why did a rule match or fail to match? The UIMA Ruta Workbench provides several views in order to answer these questions. The whole rule execution is protocolled and stored in the document represented as annotations, if the user executes the script in the debug mode.

The main view for the explanation is the *Applied Rules* view. It displays structured information about all rules that tried to match on the document (cf. Figure 3(A)). The view shows how often a rule tried to match in total and how often it succeeded. This information is given in brackets at the beginning of each rule entry. The rule

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

itself is given afterwards. Additionally, some profiling information, providing details about the absolute and relative execution time within the current block is added at the end of each rule entry.

The *Matched* view and *Failed* view display rule matches for rules selected in the *Applied Rules* view (cf. Figure 3 (B+C)). The views contain the covered text spans in order to provide a quick overview to the user. She can see, which positions have been successfully matched and which positions have been unsuccessfully investigated by the rules.

The *Rule Elements* view (D) finally displays the exact covered text for each match reference and the evaluation results for each condition. This enables the user to quickly identify the causes for a successful or failed rule match.

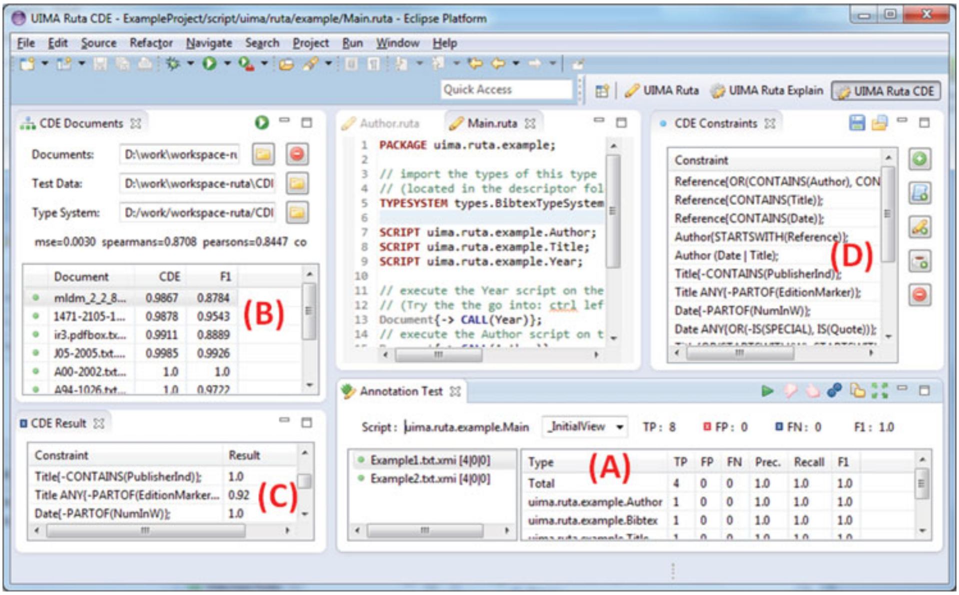
So far, the views for the explanation of the rule execution present a well-structured report if the user tries to inspect the behavior of specific rules. If, however, several rules create a certain type of annotation, then the user has to investigate each of these rules in order to identify the reasons for an erroneous annotation. The *Created By* view solves this deficiency by displaying the rule that was responsible for the creation of a specific annotation that has been selected in the *Annotation Browser* view.

The UIMA Ruta Workbench additionally provides several other views that help the user to investigate the rule execution. Among these are views that visualize the rule matches on specific positions or the rule matches of rules containing specific language elements, as well as the *Statistics* view, which provides a compact report of the execution time of the conditions and actions.

4.3 Introspection by querying

The engineered rules are not only applied on a single document during the development process, but on a collection of documents. In order to investigate the resulting annotations, the user typically has to inspect each document separately. The UIMA Ruta Workbench provides an additional view for the introspection in collections of annotated documents by querying. The *Ruta Query* view contains different text fields for specifying the folder containing the queried documents, the necessary type system and a set of UIMA Ruta rules (cf. Figure 3(E)). The view applies the rules on the collection of documents and presents the matches of the rules to the user. The rules can thus be interpreted as a query statement. The view can be applied in many scenarios during the development process. The user can, for example, obtain an overview of the occurrences of certain types of annotations or even patterns of annotations. This includes useful activities like investigating if a pattern of annotations has been or has not been annotated with a specific type. In contrast to work like Cunningham (2005) and Greenwood, Tablan, and Maynard (2011) on querying and retrieval, this view is only intended to support the knowledge engineer since no indexing is applied and the query does not scale for other applications.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press


UIMA Ruta CDE - ExampleProject/script/uima/ruta/example/Main.ruta - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
UIMA Ruta
UIMA Ruta Explain
UIMA Ruta CDE
CDE Documents
Author.ruta
Main.ruta
CDE Constraints
Annotation Test
CDE Result
(A)
(B)
(C)
(D)

Documents: D:\work\workspace-ru
Test Data: D:\work\workspace-ruta\CDI
Type System: D:/work/workspace-ruta/CDI

mse=0.0030 spearmans=0.8708 pearsons=0.8447 co

Document	CDE	F1
mldm_2_2_8...	0.9867	0.8784
1471-2105-1...	0.9878	0.9543
ir3.pdfbox.tx...	0.9911	0.8889
J05-2005.txt....	0.9985	0.9926
A00-2002.txt...	1.0	1.0
A94-1026.txt...	1.0	0.9722

Constraint	Result
Title{-CONTAINS(PublisherInd)};	1.0
Title ANY{-PARTOF(EditionMarker...	0.92
Date{-PARTOF(NumInW)};	1.0

```
PACKAGE uima.ruta.example;

// import the types of this type
// (located in the descriptor fol
TYPESYSTEM types.BibtexTypeSystem

SCRIPT uima.ruta.example.Author;
SCRIPT uima.ruta.example.Title;
SCRIPT uima.ruta.example.Year;

// execute the Year script on the
// (Try the the go into: ctrl lef
Document{-> CALL(Year)};
// execute the Author script on t
```

Constraint
Reference{OR(CONTAINS(Author), CON
Reference{CONTAINS(Title)};
Reference{CONTAINS(Date)};
Author{STARTSWITH(Reference)};
Author (Date | Title);
Title{-CONTAINS(PublisherInd)};
Title ANY{-PARTOF(EditionMarker)};
Date{-PARTOF(NumInW)};
Date ANY{OR(-IS(SPECIAL), IS(Quote))};

Script: uima.ruta.example.Main _InitialView TP: 8 FP: 0 FN: 0 F1: 1.0

- Example1.txt.xmi [4|0|0]
- Example2.txt.xmi [4|0|0]

Type	TP	FP	FN	Prec.	Recall	F1
Total	4	0	0	1.0	1.0	1.0
uima.ruta.example.Author	1	0	0	1.0	1.0	1.0
uima.ruta.example.Bibtex	1	0	0	1.0	1.0	1.0
uima.ruta.example.Title	1	0	0	1.0	1.0	1.0

Fig. 4. (Colour online) The UIMA Ruta Workbench: Different views for automatic validation. (A) The *Annotation Testing* view for gold standard evaluation. (B) Results of constraint-driven evaluation. (C) Detailed results of specific constraints. (D) Collection of expectations.

4.4 Automatic validation

The manual validation of the results of a currently developed rule set can be tedious and time consuming, especially if several documents have to be inspected. The user has to perform this task potentially each time the rule set is extended or refined in order to ensure the correct processing of the documents. Hence, an automatic validation of the rule set's correctness is one of the most important features of a development environment for rule-based information extraction. The UIMA Ruta Workbench provides tools for the automatic validation of rule sets using labeled and unlabeled documents.

4.4.1 Gold standard evaluation

The user should be able to define test cases that are utilized to validate the rule set. In the context of information extraction applications, the test cases are equivalent to a set of documents containing annotations of interesting types. The rules are applied on the raw documents and the resulting annotations created by the rules are then compared to the given gold annotations. The differences are used to calculate an evaluation score, e.g., the F_1 score.

The UIMA Ruta Workbench provides this functionality with the *Annotation Testing* view. Figure 4(A) contains a screenshot of this view, which consists of the list of tested documents on the left side and the results for the currently selected document on the right side. Additionally, the user is able to select different evaluators resulting in different $F1$ scores, e.g., based on annotations, tokens or features. The

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

usefulness of this functionality can be summarized with the description of three use cases:

Goal-oriented development A real-world scenario for the development of a rule-based information extraction system often includes a quality threshold specified by a contractee. Given a set of annotated documents, the developed rules have to achieve a previously defined evaluation score in order to be accepted for deployment. This use case is directly supported by the UIMA Ruta Workbench and the user is able to continuously compare the current state of the application to the desired targets.

Test-driven development Test-driven development (Janzen and Saiedian 2005) is a programming process model where first test cases for specific parts of functionality are defined before the actual program code is written. The developed code is then continuously validated during development in order to ensure the correctness of different parts of the software. Test-driven development has proven itself as an effective process model (Maximilien and Williams 2003). This process model can also be applied to the development of rule-based information extractions systems. First, a set of documents is manually annotated, which provides a best possible coverage of different challenges. The rules are then developed against these test cases until the performance of the rules is sufficient. A methodology and detailed process model for test-driven development of rule-based information extraction systems has been published by Kluegl, Atzmueller, and Puppe (2009b).

Regression testing Another reliable process model for developing rules can be summarized as follows. The rule engineer considers one document after each other and creates new rules, or refines old rules. Starting with the first document, she creates an initial set of rules, which extracts interesting information within this document. After the rules are applied on the document, it is either perfectly annotated or manually corrected, and is stored as a test case. The rule engineer continues with the second document and extends, modifies and refactors the rule set until the annotations in the second document are correctly identified. During these modifications of the rules set, however, the test case of the first document is continuously validated in order to ensure that the rules still provide the necessary functionality for the first document. This procedure is iterated for the complete collection of documents until all documents are sufficiently processed by the created rules.

4.4.2 Constraint-driven evaluation

One of the advantages of rule-based information extraction approaches is that annotated documents are not strictly necessary for application development. Nevertheless, they are very beneficial for rule-based systems, e.g., for accelerating the development or quality maintenance. There is a natural lack of labeled data in most application domains and its creation is error-prone, cumbersome and time-consuming as is the manual validation of the extraction results by a human. A human is able to validate the created annotations in those documents using background knowledge

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

and expectations on the domain. An automatic estimation of the rule set's quality in unseen and unlabeled documents using this kind of knowledge provides many advantages and greatly improves the engineering experience.

The Constraint-driven Evaluation (CDE) framework (Wittek *et al.* 2013) is a combination of the testing framework and the querying functionality, and greatly improves the engineering process in the opinion of the authors. It allows the user to specify expectations about the domain in form of constraints. These constraints are applied on documents with annotations, which have been created by a set of rules. The results of the constraints are aggregated to a single CDE score, which reflects how well the created annotations fulfill the user's expectations and thus provide a predicted measurement of the rule set's quality for these unlabeled documents. The documents can be ranked according to the CDE score, which provides an intelligent report about the well and poorly processed examples. Figure 4 (B+C+D) provides a screenshot of different views to formalize the set of constraints and to present the predicted quality of the model for the specified documents.

Compared to the test-driven development in the last section, this approach facilitates a constraint-driven development, which requires no annotated data. This process is illustrated with an simple example for identifying one specific person name in each document of a larger collection. The knowledge engineer specifies her background knowledge and expectations about the domain using rules. These rules cover, for example, that each document should contain exactly one annotation of the type Name. After the actual extraction rules are applied on the large set of documents, the expectations are compared to the Name annotations created by the rules. Using the CDE score, the documents are ranked where documents that violate the expectations are listed first. The knowledge engineer is now able to investigate these problematic documents where the rule obviously failed either by finding no name or by labeling multiple names. After the rule set is refined or extended, this process is iterated. More information about this tool can be found in Wittek *et al.* (2013).

4.5 Supervised rule induction

All development support described until now has focused on the manual engineering of rule-based information extraction systems. The user defines the set of rules in an ecosystem of tools, which facilitate the writing or enable an improved quality maintenance. Besides that, the development environment can also support the user in the construction of new rules. Given a set of annotated documents, machine learning algorithms can be applied in a supervised fashion in order to propose new rules. The user can then inspect the proposed rules for insights in possibly interesting patterns of annotations. Furthermore, she can extend her rule set with a selection of the proposed rules, which can again be extended or adapted.

The UIMA Ruta Workbench provides the *TextRuler* framework for the supervised induction of rules. The *TextRuler* view allows the user to specify the training data, the interesting types of annotations and the preferred learning algorithm. The induced

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

rules for each algorithm are presented in a separate view. Currently, four different algorithms are available.

LP^2 The two implementations of this rule learner, naive and optimized, are adaptations of the original algorithm published by Ciravegna (2003). LP^2 induces three different kinds of rules. Tagging rules identify the boundaries of the annotations, context rules shift misplaced boundaries and correction rules finally are able to remove boundaries again. Correction rules are, however, not yet supported by our implementations.

WHISK The two implementations of this rule learner, token and generic, are adaptations of the algorithm published by Soderland, Cardie, and Mooney (1999). The Whisk algorithm induces rules in the form of modified regular expressions. In contrast to the original algorithm, our implementations do not directly support multi-slot rules.

KEP The name of the rule learner KEP (knowledge engineering patterns) is derived from the idea that humans use different engineering patterns to write annotation rules. Our algorithm implements simple rule induction methods for some patterns, such as boundary detection or annotation-based restriction of the context. The results are then combined in order to take advantage of the interaction of different kinds of induced rules. Since the single rules are constructed according to how humans engineer rules, the resulting rule set resembles more handcrafted rule sets. Furthermore, by exploiting synergy of patterns, the patterns for some annotations are much simpler.

TraBaL Our TraBaL algorithm (Eckstein, Kluegl and Puppe 2011) is able to induce transformation-based error-driven rules. The basic idea is similar to the Brill Tagger (1995), but the template generation is more generic and can also handle arbitrary annotations instead of tags of tokens. This algorithm was built in order to learn how to correct the annotations of arbitrary models or human annotators.

A methodology and process model for the semi-automatic development of rule-based information extraction systems where the human engineer is supported and completed by rule induction algorithms has been published in Kluegl *et al.* (2009a).

4.6 Semi-automatic creation of gold documents

A popular approach for creating annotated documents is the semi-automatic annotation using rules. The labeling of a large collection of documents is time consuming, but can be accelerated if recurring annotations are automatically created. The rule engineer defines a few rules that are able to create correct annotations instead of manually specifying them one after each other. Afterwards, the created annotations need to be verified, whereas missing annotations are added and defective annotations are removed. This functionality is already covered by the default tooling of the UIMA Workbench in combination with the CAS Editor. The approach can, however, be improved if the user is supported in the manual verification of the annotations. The UIMA Ruta Workbench provides the additional view *Check*

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

```
Macro: AMOUNT_NUMBER
({Token.kind == number}
 (({Token.string == ","} |
   {Token.string == "."})
  {Token.kind == number})*
)
Rule: Money1
(
  (AMOUNT_NUMBER)
  (SpaceToken.kind == space)?
  ({Lookup.majorType == currency_unit})
)
:money  -->
  :money.Number = {kind = "money", rule = "Money1"}
```

```
(NUM (("," | ".") NUM)*)
    {-> AmountNumber};
(AmountNumber SPACE? CurrencyUnit)
    {-> Money};
```

Fig. 5. An excerpt of an exemplary JAPE macro and rule (Cunningham *et al.* 2000) (left) for the detection of 'money' entities and their UIMA Ruta equivalents (right).

Annotations for this use case, which enables the user to efficiently accept, reject or replace the proposed annotations. The view lists all annotations of types selected by the user, which can quickly classify these annotations as correct or erroneous.

In summary, the UIMA Ruta Workbench provides a large amount of useful tooling that helps to improve the engineering experience and lowers the overall development time and costs.

5 Comparison to related systems

We compare UIMA Ruta to a representative selection of related systems and highlight different aspects of rule representation and execution, expressiveness of the language, runtime performance, and available tooling for development support. A special focus is laid on the concise representation of rules, which is one important aspect for rapid development. The less text the knowledge engineer has to write for achieving the same functionality, the better. The compactness and expressiveness of the UIMA Ruta language is illustrated in Figures 5, 7, and 8. Each figure depicts a representative example of a related language taken from the respective publication and its equivalent in the UIMA Ruta language.

Figure 5 contains an example of a JAPE macro and rule (Cunningham *et al.* 2000) and their equivalents in UIMA Ruta. UIMA Ruta does not enforce a clear separation of conditions and actions and thus does not need to support labels. Java code cannot directly be included in UIMA Ruta rules, but the language itself can be extended and arbitrary Java code wrapped in additional analysis engines can be executed. This integration of functionality implemented in Java is, however, more complex compared to JAPE, but allows tooling like explanation and editing support. JAPE specifies the accessible types for each phase, whereas UIMA Ruta applies a more complex and dynamic paradigm of coverage-based visibility controlled by the annotations themselves. In contrast to JAPE, which compiles all rules of a phase into one FST, UIMA Ruta applies the rules sequentially in the order they are specified, supports variable matching direction, and is able to match on all disjunctive alternatives. Overall, the UIMA Ruta language provides almost all features of JAPE together with a more concise representation. The development of JAPE grammars is barely supported by tooling to the best knowledge of the authors. The GATE

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

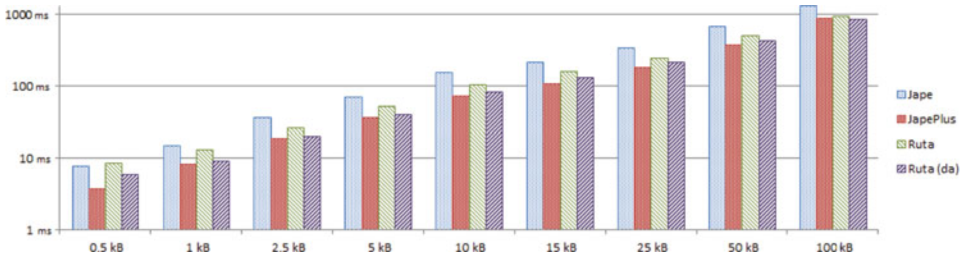


Fig. 6. (Colour online) Average processing time for documents of different sizes.

framework provides, however, a rich selection of tools like ANNIC (Cunningham 2005) for discovering new patterns.

For a comparison of the runtime performance, we utilized the Named Entity Recognition component in GATE without the part-of-speech tagger and applied only the gazetteers and the first three phases of the JAPE grammar with overall 58 rules. A manual translation of these rules to UIMA Ruta resulted in about 44 rules, which can be considered as created by an inexperienced knowledge engineer. They do not include UIMA Ruta specific optimizations. Both systems are applied on nine sets of documents each containing 10,000 documents and providing an increasing document size, taken from the Enron email dataset[9]. Startup time of the frameworks and Java can be neglected, because we evaluated the first batch twice, but only measured the second run. Figure 6 depicts the average runtime for components based on the JAPE implementation, the optimized JAPE Plus implementation[10], UIMA Ruta and UIMA Ruta with activated dynamic anchoring[11]. UIMA Ruta is able to compete well for all sizes of documents, although the rules are not optimized at all, and they sequentially apply 44 phases in contrast to three phases of JAPE. Dynamic anchoring improves the performance only slightly since the patterns have not been engineered accordingly and considerable processing time is caused by the gazetteers.

The UIMA Ruta language provides a higher expressiveness than AFST (Boguraev *et al.* 2010), which is limited to a linear path through the annotation lattices. Each special functionality in AFST (honor, focus, advance, . . .) is available in UIMA Ruta using the corresponding language constructs. Figure 7, for example, contains exemplary rules for vertical navigation in AFST and UIMA Ruta. The AFST rule starts by matching on a PName annotation and then steps into this annotation indicated by the operator '@descend'. The next element 'Title[string=='General']' specifies that the PName annotation has to start with a Title annotation with the covered text 'General'. A period between two elements indicates a sequential constraint. The vertical navigation is repeated for the Name annotation, which has to contain a Last annotation with the covered text 'Grant'. The elements First and Middle are optional, but required for the linear path through the annotation lattices.

[9] https://www.cs.cmu.edu/ enron/
[10] http://gate.ac.uk/sale/tao/splitch8.html#sec:jape:plus
[11] We applied GATE 7.1 and UIMA Ruta 2.2.0.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

```
findG = PName[@descend] .
          Title[string=="General"] .
          Name[@descend] .
            First[]|<E> . Middle[]|<E>
            . Last[string=="Grant"] .
          Name[@ascend] .
        PName[@ascend] ;
```

```
PName<-{
  Title{REGEXP("General")}
  Name<-{
    Last{REGEXP("Grant")};
  };
};
```

Fig. 7. (Colour online) An exemplary AFST rule (Boguraev *et al.* 2010) (left) for vertical matching in ‘PName’ annotations and its UIMA Ruta equivalent (right). The rules match on text passages like ‘General Ulysses S. Grant’ if the corresponding annotations are present. The optional patterns for the First and Middle annotations are not necessary in UIMA Ruta.

```
create view CapsLast as
select  CombineSpans(C.name, L.name) as name
from    Caps C, Last L
where   FollowTok(C.name, L.name, 0 0);
...
create view PersonAll as
  (select R.name from FirstLast R) union all ...
  ... union all (select R.name from CapsLast R);

create view Person as select * from PersonAll R
consolidate on R.name using ’ContainedWithin’;
```

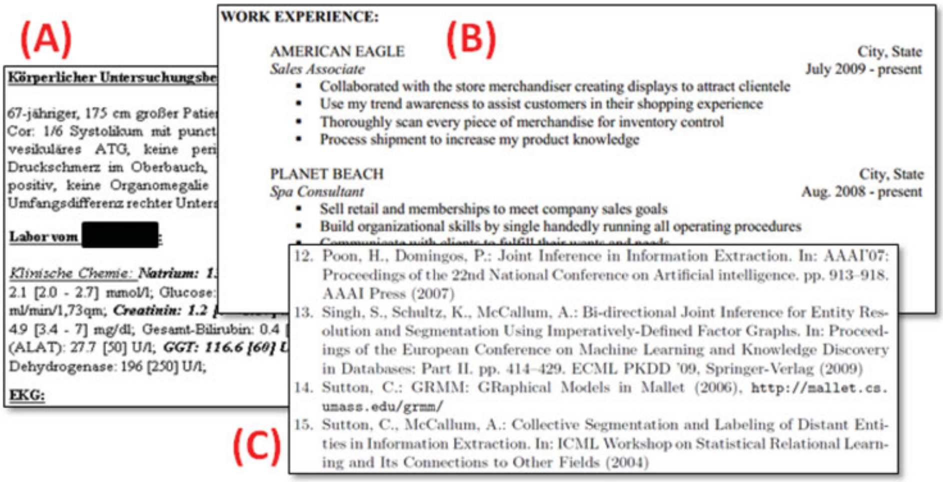
```
(Caps Last){-> Person};
Person{PARTOFNEQ(Person)
  -> UNMARK(Person)};
Person{CONTAINS(Person,2,100)
  -> UNMARK(Person)}
```

Fig. 8. (Colour online) Excerpt of exemplary AQL rules (Chiticariu *et al.* 2010) (left) for the detection of persons and their UIMA Ruta equivalents (right). The last two UIMA Ruta rules are only necessary for the consolidate statement.

AFST includes a small set of additional predicates, whereas UIMA Ruta ships an extensive set of conditions and actions. The Domain Adaptation Toolkit (Boguraev and Neff 2006) provides grammar development functionality and is able to create type system descriptors based on the grammars like the UIMA Ruta Workbench.

SystemT criticizes three aspects of rule languages based on the CPSL specification: Lossy sequencing, rigid matching priority and limited expressiveness in rule patterns (Chiticariu *et al.* 2010). None of these properties can be observed in UIMA Ruta. The rule language of SystemT, AQL, is a declarative relational language similar to SQL and thus does not provide a compact representation. Especially since the modification of the content of a view enforces the specification of another view. While the syntax is accessible to programmers, it might appear counterintuitive for users not familiar with SQL. Most of the features of AQL are also supported in the UIMA Ruta language. Figure 8 provides an excerpt of an AQL grammar for the detection of persons and the UIMA Ruta rules with the same functionality. Broadly speaking, AQL rules typically consist of a `create view` statement that specifies the created type of annotation, `select` and `from` statement for specifying the input and output, and a `where` statement for the pattern. The first UIMA Ruta rule is equivalent to the first `create view` statement in the AQL example. There is no need for the second `create view` statement in UIMA Ruta since the rules are able to operate on the same types of annotations. The second and third UIMA Ruta rules emulate the last `create view` statement, which consolidates overlapping annotations.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press



Körperlicher Untersuchungsbe

67-jähriger, 175 cm großer Patie
Cor: 1/6 Systolikum mit punct
vesikuläres ATG, keine peri
Druckschmerz im Oberbauch,
positiv, keine Organomegalie
Umfangsdifferenz rechter Unters

Labor vom [redacted]:

Klinische Chemie: ***Natrium: 1***
2.1 [2.0 - 2.7] mmol/l; Glucose:
ml/min/1,73qm; ***Creatinin: 1.2***
4.9 [3.4 - 7] mg/dl; Gesamt-Bilirubin: 0.4
(ALAT): 27.7 [50] U/l; ***GGT: 116.6 [60] U***
Dehydrogenase: 196 [250] U/l;

EKG:

WORK EXPERIENCE:

AMERICAN EAGLE — City, State
Sales Associate — July 2009 - present

- Collaborated with the store merchandiser creating displays to attract clientele
- Use my trend awareness to assist customers in their shopping experience
- Thoroughly scan every piece of merchandise for inventory control
- Process shipment to increase my product knowledge

PLANET BEACH — City, State
Spa Consultant — Aug. 2008 - present

- Sell retail and memberships to meet company sales goals
- Build organizational skills by single handedly running all operating procedures
- Communicate with clients to fulfill their wants and needs

12. Poon, H., Domingos, P.: Joint Inference in Information Extraction. In: AAAI'07: Proceedings of the 22nd National Conference on Artificial intelligence. pp. 913–918. AAAI Press (2007)
13. Singh, S., Schultz, K., McCallum, A.: Bi-directional Joint Inference for Entity Resolution and Segmentation Using Imperatively-Defined Factor Graphs. In: Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part II. pp. 414–429. ECML PKDD '09, Springer-Verlag (2009)
14. Sutton, C.: GRMM: GRaphical Models in Mallet (2006), `http://mallet.cs.umass.edu/grmm/`
15. Sutton, C., McCallum, A.: Collective Segmentation and Labeling of Distant Entities in Information Extraction. In: ICML Workshop on Statistical Relational Learning and Its Connections to Other Fields (2004)

Fig. 9. (Colour online) Excerpts of exemplary documents processed in the case studies: German clinical letters (A), curricula vitae (B), and scientific references (C).

The rule execution of SystemT is not based on FSTs, but applies an optimized operator plan for the execution of rules. The rules in UIMA Ruta are also not limited to a left-to-right matching, which can greatly improve the runtime performance. The automatic selection of the starting rule element (dynamic anchoring) is a first step toward an optimized execution plan. UIMA Ruta was developed for the rapid development of rules and cannot (yet) compete with SystemT concerning runtime performance.

SystemT provides the best tools for development support of all related systems, to the best knowledge of the authors. Most of the features are also supported in a similar form by the UIMA Ruta Workbench. The development support of UIMA Ruta provides more possibilities to automatically estimate the quality of the rules, e.g., also on unlabeled documents, which is an essential assistance for developing rules. Another development environment for creating rules is the IBM Content Analytics Studio, which propagates a drag-and-drop paradigm for specifying patterns instead of a textual language. In contrast to UIMA Ruta, the system provides a more sophisticated dictionary support, but lacks many advantages of flexible rule languages. In our experience, trained knowledge engineers are faster in specifying rule sets in a textual form.

6 Case studies

This section describes a selection of information extraction applications implemented by the authors of this paper with UIMA Ruta. All case studies process domains with semi-structured texts. Examples of these documents are given in Figure 9. Table 1 provides an overview of the case studies' key figures. UIMA Ruta is not restricted to these kinds of documents, but can also be applied for structured documents, free texts or other tasks.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

Table 1. *Key figures of case studies: amount of involved rules, effort spent for rule development, size (#documents) of development and test set, and F_1 score on unseen test set. *Confidential project with industrial partner where effort and F_1 score are not available, but the company confirmed an increase of efficiency by 100%*

		#rules	effort	#documents (dev/test)	F_1 score (test set)
Curricula vitae	(Section 6.1)	25	1 h	5/10	0.979
Clinical letters	(Section 6.1)	102	<2 h	15/126	0.972
References	(Section 6.2)	1884	<6 h	12/21	0.997
Curricula vitae	(Section 6.3)	≈1000	n/a	n/a	n/a*
Clinical letters	(Section 6.4)	463	n/a	500/200	0.992

6.1 Increasing recall in precision-driven prototyping

The two case studies published in Kluegl, Atzmueller, and Puppe (2009c) are both proof-of-concept prototypes using UIMA Ruta. They apply precision-driven rules, which are able to identify confident information, and then find additional information using the regularities in the document. The first case study identifies companies in English curricula vitae with dictionaries and some simple rules for projection of consistent compositions (cf. Figure 9(B)). About 25 rules have been specified for this task without any specific process model. The second case study detects headlines in clinical letters for further segmentation and classification of their contents (cf. Figure 9(A)). The rules first extract confident headlines and then find the remaining headlines using the layout of the known headlines (cf. Section 6.4). Overall, the rule set contains 102 rules. The applied datasets for the development and evaluation of the prototypes consist of 15 curricula vitae with 72 companies and 141 letters with 1515 headlines whereas thirty and ten percent have been utilized as a development set. Both rule sets achieve an F_1 score of over 0.97 on the remaining unseen documents and have been created in less than 2 h.

6.2 Segmentation of references

The rules in the case study of Kluegl, Hotho, and Puppe (2010) have been engineered to extract the BibTeX entities *Author*, *Title*, *Editor*, and *Date/Year* in references of scientific publications (cf. Figure 9(C)). The approach is implemented with three sets of rules, which have been engineered using 223 labeled references in 12 and a combination of test-driven development and regression testing. The first set of rules serves as a simple base component, which already extracts the targeted entries, but was created with minimal effort. The second set analyzes the boundaries of the entries and creates a model of the consistencies of the document by investigating the dominant composition of entities. The third set finally consists of transformation rules, which change the type and offsets of the entries dependent on the model of local consistencies. Overall, the application consists of 1884 rules, which have been engineered in less than 6 h. The approach achieves remarkable results in the limited scenario of 299 unseen and well-formatted references in 21 documents with

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

an average F_1 score of 0.997. Processing a reference section with 20 references takes in average 0.418 seconds. The F_1 score of the base component is 0.983. Example 18 contains a transformation-based rule that corrects Author annotations dependent on the consistencies.

```
Author->{#{-> SHIFT(Author,1,2)} EndOfAuthor ANY[0,4]? ConflictAtEnd;};
```

Example 18. Example of a transformation-based rule for correcting Author annotations dependent on a type variable (EndOfAuthor), which stores the dominant ending of authors, and an additional annotation (ConflictAtEnd) that points out discrepancies to the local model.

6.3 Template extraction in curricula vitae

UIMA Ruta was applied for template extraction in German curricula vitae (cf. Figure 9(B)). An early version is described in Atzmueller, Kluegl, and Puppe (2008). The goal was to detect the projects including the start and end date, the employer, the title and the important skills. The application utilized rules that imitate the perception of humans in order to identify the project segments. This is achieved by modeling different layers of perceptible text fragments (paragraph, table, line) and finding minimal structures with multiple dates and maximal structures with exactly one date. This process is extended with patterns for repetitive fragments. Then, extensive dictionaries and contextual rules are applied for the extraction of interesting information within these segments. The application consists of about 1,000 rules engineered using a methodology similar to regression testing, but without a labeled dataset. It was still necessary to manually validate the extraction results, but the application was able to double the efficiency of the division that populates databases with the extracted information.

6.4 Segmentation of clinical letters by headline identification

The application described in Beck (2013) provides a real-world implementation for the segmentation of clinical letters mentioned in Section 6.1 (cf. Figure 9(A)). The goal is the recognition and classification of different sections, such as diagnosis, history, various kinds of examinations, therapy and epicrisis. These sections are then processed by further information extraction applications in order to populate a clinical data warehouse. The rules have been engineered using 500 labeled letters using the test-driven development methodology. The script files consist overall of 463 rules and 217 other statements for blocks, declarations and imports. First, the rules extract the headlines in the document and then use the headlines for the identification and categorization of the sections. Headlines are detected using different approaches based on the formatting, semantic keywords, or consistencies within the letter. By combining these approaches, the application is able to achieve an F_1 score of 0.993 on 200 unseen documents. Our baseline similar to a component of cTAKES (Savova *et al.* 2010) using only keywords and regular expressions resulted in an F_1 score of 0.912.

7 Conclusions

UIMA Ruta is a useful tool for rule-based information extraction in the ecosystem of UIMA. The system was designed with a special focus on rapid development in order to reduce development time and costs. The rule language can be applied for solving various use cases, but still provides a compact representation. It covers most functionality of related languages and still introduces a few new features that ease the specification of complex patterns. The UIMA Ruta Workbench adds another important aspect for rendering rapid development possible. It provides extensive tooling support for all tasks that a knowledge engineer has to perform when creating rule-based information extraction applications. UIMA Ruta is currently unique concerning the combination of integration in UIMA, expressiveness of its language and industry-friendly open source license.

The future work in the development of UIMA Ruta is mainly driven by the requirements of the community. In order to establish the system further and to improve its acceptance in the community, more examples and ready-to-use rule sets will be provided in the future. Since UIMA Ruta has put its focus on rapid development, there remain many opportunities to enhance the runtime performance. Two directions can be identified: the introduction of language elements that compile multiple rules into one automaton and the intensified usage of optimized execution plans. Even if the language specification of UIMA Ruta is already compact and expressive, it still can be extended to cover different use cases more efficiently. One example is the incorporation of unification-based techniques for coreference tasks. Java code directly embedded in rules would allow experienced users to close gaps of functionality with the cost of reduced tooling support. Ontologies are not yet directly supported, but can improve information extraction applications. Finally, advanced rule induction algorithms and similar tools will be able to support the knowledge engineer further in the creation of information extraction applications.

Acknowledgments

This work was supported by the Competence Network Heart Failure, funded by the German Federal Ministry of Education and Research (BMBF01 EO1004).

References

Appelt, D. E., and Onyshkevych, B. 1998. The common pattern specification language. In *Proceedings of a Workshop on Held at Baltimore, Maryland: October 13–15, 1998, (TIPSTER '98)*, Stroudsburg: ACL, pp. 23–30.

Atzmueller, M., Kluegl, P., and Puppe, F. 2008. Rule-based information extraction for structured data acquisition using TextMarker. In Baumeister and Atzmueller (ed.) *LWA-2008 (Special Track on Knowledge Discovery and Machine Learning)*, Würzburg, Germany, pp. 1–7.

Beck, P.-D. 2013. *Identifikation und Klassifikation von Abschnitten in Arztbriefen (in German)*. Master Thesis, University of Würzburg.

Black, W. J., McNaught, J., Vasilakopoulos, A., Zervanou, K., Theodoulidis, B., and Rinaldi, F. 2005. CAFETIERE: conceptual annotations for facts, events, terms, individual entities and RElations. Technical Report TR–U4.3.1. Parmenides Technical Report.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

Boguraev, B., and Neff, M. 2006. An annotation-Based finite-state system for UIMA: pattern matching over annotations. Technical Report, IBM T.J. Watson Research Center.

Boguraev, B., and Neff, M. 2010. A framework for traversing dense annotation lattices. *Language Resources and Evaluation* **44**(3): 183–203.

Bohannon, P., Merugu, S., Yu, C., Agarwal, V., DeRose, P., Iyer, A., Jain, A., Kakade, V., Muralidharan, M., Ramakrishnan, R., and Shen, W. 2009. Purple SOX extraction management System. *SIGMOD Record* **37**(4): 21–27

Brill, E. 1995. Transformation-based error-driven learning and natural language processing: a case study in part-of-speech tagging. *Computational Linguistics* **21**(4): 543–565.

Chiticariu, L., Chu, V., Dasgupta, S., Goetz, T. W., Ho, H., Krishnamurthy, R., Lang, A., Li, Y., Liu, B., Raghavan, S., Reiss, F. R., Vaithyanathan, S., and Zhu, H. 2011. The systemT IDE: an integrated development environment for information Extraction rules. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, New York: ACM, pp. 1291–1294.

Chiticariu, L., Krishnamurthy, R., Li, Y., Raghavan, S., Reiss, F. R., and Vaithyanathan, S. 2010. SystemT: an algebraic Approach to declarative information extraction. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, Stroudsburg: ACL, pp. 128–137.

Chiticariu, L., Li, Y., and Reiss, F. R. 2013. Rule-based information extraction is dead! Long live rule-based information extraction systems! In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, Stroudsburg: ACL, pp. 827–832.

Ciravegna, F. 2003. (LP)2, rule induction for information extraction using linguistic constraints. Technical Report CS–03–07, Department of Computer Science, University of Sheffield.

Cunningham, H. 2007. Indexing and querying linguistic metadata and document content. In *Recent Advances in Natural Language Processing IV: Selected papers from RANLP 2005*. 292, pp. 35–44. Amsterdam: John Benjamins Publishing Company.

Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V., Aswani, N., Roberts, I., Gorrell, G., Funk, A., Roberts, A., Damljanovic, D., Heitz, T., Greenwood, M. A., Saggion, H., Petrak, J., Li, Y., and Peters, W. 2011. *Text Processing with GATE (Version 6)*. Murphys, CA: Gateway Press.

Cunningham, H., Maynard, D., and Tablan, V. 2000. JAPE: a java annotation patterns engine (Second Edition). Research Memorandum CS–00–10, Department of Computer Science, University of Sheffield, Sheffield.

David J., and Hossein S. 2005. Test-driven development: concepts, taxonomy, and future direction. *Computer* **38**(9): 43–50.

Doan, A., Granavo, L., Ramakrishnan, R., and Vaithyanathan, S. 2008. *Special Issue on Managing Information Extraction*. New York: ACM.

Drozdzynski, W., Krieger, H.-U., Piskorski, J., Schäfer, U., and Xu, F. 2004. Shallow processing with unification and typed feature structures - foundations and applications. *Künstliche Intelligenz* **18**(1): 17–23.

Eckstein, B, Kluegl, P., and Puppe, F. 2011. Towards learning error-driven transformations for information extraction. In *Workshop Notes of the LWA 2011 - Learning, Knowledge, Adaptation*, Magdeburg, Germany, pp. 199–204.

Fagin, R., Kimelfeld, B., Reiss, F., and Vansummeren, S. 2013. Spanners: a formal framework for information extraction. In *Proceedings of the 32nd Symposium on Principles of Database Systems*, New York: ACM, pp. 37–48.

Ferrucci, D., and Lally, A. 2004. UIMA: An architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering* **10**(3/4): 327–348.

Ferrucci, D. A., Brown, E. W., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A., Lally, A., Murdock, J. W., Nyberg, E., Prager, J. M., Schlaefer, N., and Welty, C. A. 2010. Building Watson: an overview of the DeepQA project. *AI Magazine* **31**(3): 59–79.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

Greenwood, M. A., Tablan, V., and Maynard, D. 2011. GATE Mmir: answering questions Google Cant. In *Proceedings of the 10th International Semantic Web Conference (ISWC2011). Lecture Notes in Computer Science*, vol. 7031. Springer.

Gurevych, I., Mühlhäuser, M., Müller, C., Steimle, J., Weimer, M., and Zesch, T. 2007. Darmstadt knowledge processing repository based on UIMA. In *Proceedings of the First Workshop on Unstructured Information Management Architecture at Biannual Conference of the Society for Computational Linguistics and Language Technology*, Heidelberg: Springer.

IJntema, W., Sangers, J., Hogenboom, F., and Frasincar, F. 2012. A lexicosemantic pattern language for learning ontology instances from text. *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 15. Amsterdam: Elsevier, pp. 37–50.

Khaitan, S., Ramakrishnan, G., Joshi, S., and Chalamalla, A. 2008. RAD: a scalable framework for annotator development. In Alonso, Blakeley and Chen (ed.), *ICDE*, Los Alamitos, CA: IEEE Computer Society Press, pp. 1624–1627.

Kluegl, P., Atzmueller, M., Hermann, T., and Puppe, F. 2009a. A framework for semi-automatic development of rule-based information extraction applications. In Hartmann and Janssen (ed.), *Proceedings LWA 2009 (KDML - Special Track on Knowledge Discovery and Machine Learning)*, Darmstadt, Germany, pp. 56–59.

Kluegl, P., Atzmueller, M., and Puppe, F. 2009b. Test-driven development of complex information Extraction systems using TextMarker. In Naplepa and Baumeister (ed.), *4th International Workshop on Knowledge Engineering and Software Engineering (KESE 2008), 31th German Conference on Artificial Intelligence (KI-2008)*, Darmstadt, Germany, pp. 19–30.

Kluegl, P., Atzmueller, M., and Puppe, F. 2009c. Meta-level information extraction. In *32nd Annual German Conference on Artificial Intelligence (KI 2009)*, Berlin: Springer, pp. 233–240.

Kluegl, P., Atzmueller, M., and Puppe, F. 2009d. TextMarker: a tool for rule-based information Extraction. In Chiarcos, Eckart de Castilho and Stede (ed.), *Proceedings of the Biennial GSCL Conference 2009, 2nd UIMA@GSCL Workshop*, Tübingen: Gunter Narr Verlag, pp. 233–240.

Kluegl, P., Hotho, A., and Puppe, F. 2010. Local adaptive extraction of references. In *33rd Annual German Conference on Artificial Intelligence (KI 2010)*, Berlin: Springer, pp 40–47.

Lafferty, J., McCallum, A., and Pereira, F. 2001. Conditional random fields: probabilistic models for segmenting and labeling sequence data. In *Proceedings 18th International Conference on Machine Learning*, San Francisco: Morgan Kaufmann, pp. 282–289.

Li, Y., Chiticariu, L., Yang, H., Reiss, F. R., and Carreno-fuentes, A. 2012. WizIE: a best practices guided development environment for information extraction. In *Proceedings of the ACL 2012 System Demonstrations, ACL '12*, Stroudsburg: ACL, pp. 109–114.

Maximilien, E. M., and Williams, L 2003. Assessing test-driven development at IBM. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, Los Alamitos: IEEE Computer Society Press, pp. 564–569.

Piskorski, J., and Yangarber, R. 2013. Information extraction: past, present and future. In Poibeau, Saggion, Piskorski and Yangarber (ed.), *Multi-source, Multilingual Information Extraction and Summarization, Theory and Applications of Natural Language Processing*, Berlin: Springer, pp. 23–49.

Ramakrishnan, G., Balakrishnan, S., and Joshi, S. 2006. Entity Annotation based on inverse index operations. In Jurafsky and Gaussier (ed.), *EMNLP*, Stroudsburg: ACL, pp. 492–500.

Savova, G. K., Masanz, J. J., Ogren, P. V., Zheng, J., Sohn, S., Kipper-Schuler, K. C., and Chute, C. G. 2010. Mayo clinical text analysis and knowledge extraction system (cTAKES): architecture, component evaluation and applications. *Journal of the American Medical Informatics Association: JAMIA* **17**(5): 507–513.

Shen, W., Doan, A., Naughton, J. F., and Ramakrishnan, R. 2007. Declarative information extraction using datalog with embedded extraction predicates. In *Proceedings of the 33rd*

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press

International Conference on Very Large Data Bases, (VLDB '07), VLDB Endowment, pp. 1033–1044.

Soderland, S., Cardie, C., and Mooney, R. 1999. Learning information extraction rules for semi-structured and free text. *Machine Learning* **34**: 233–272.

Turmo, J., Ageno, A., and Català, N. 2006. Adaptive information extraction. *ACM Computing Surveys* **38**(2): 1–47.

Wittek, A., Toepfer, M., Fette, G., Kluegl, P., and Puppe, F. 2013. Constraint-driven evaluation in UIMA Ruta. In Kluegl, Eckart de Castilho and Tomanek (ed.), *UIMA@GSCL, CEUR Workshop Proceedings*, vol. 1038. CEUR-WS.org, pp. 58–65.

Yang, H., Pupons-Wickham, D., Chiticariu, L., Li, Y., Nguyen, B., and Carreno-Fuentes, A 2013. I can do text analytics!: Designing development tools for novice developers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, (CHI '13)*, New York: ACM, pp. 1599–1608.

https://doi.org/10.1017/S1351324914000114 Published online by Cambridge University Press