Single Rule Evaluation (SRE): Computational Algorithmic Debugging for Complex SWRL Rules

Jannik Geyer, Johannes Nguyen, Thomas Farrenkopf and Michael Guckert
KITE, Technische Hochschule Mittelhessen, Wilhelm-Leuschner-Straße 13, Friedberg, Germany

Keywords: Rule Evaluation, SWRL, OWL, Ontology Debugging.

Abstract: SWRL is an extension for OWL which allows the use of Horn clause like rules in ontologies. SWRL rules are an expressive instrument for OWL-based ontologies simplifying and augmenting deductive reasoning capabilities. With increasing size and complexity rule bases becomes more and more fragile as logical inconsistencies in the overall structure of the rule base are difficult to find. However, available debugging options require immense manual effort, if not even become an impossible task. Therefore, there is an expressed need for developers and end users to get an efficient and easy to use interactive rule evaluation instrument. In this paper we present a new method for a simplified debugging process that we call *Single Rule Evaluation (SRE)*. This SRE method enables the user to iterate through the reasoning process of the ontology and the set of inference rules and examines each atom of a selected SWRL Rule to deliver detailed information about the inferred output. In addition to a theoretical concept, we present a prototypical implementation of SRE as a *Protégé* plugin that can be invoked during the modelling process to test rules for consistency.

1 INTRODUCTION

SWRL (Semantic Web Rule Language) is a rule language based on RuleML extending OWL (Web Ontology Language) with Horn clause like if-then rules (Horrocks et al., 2004). With a growing number of developers the need for appropriate debugging tools for SWRL rules in large knowledge bases becomes more prominent as was already stated by the creator of *Protégé* Martin O'Connor himself in a blog post (O'Connor, 2018). Up to now there are only few open source solutions for that purpose and only a single commercial product called *ODASE Rules Workbench* (MacLarty et al., 2016). However, it should be noted that most of the available solutions only offer an insufficient scope of debugging functionalities. An example is the Fluent Editor 2015, which is able to display all SWRL rules together with all relevant elements from the ontology executed by the Rule Engine (flu,). Nonetheless, information about elements that caused a rule to fail are missing. Computer-aided debugging methods reduce manual efforts and increase the efficiency of the development process. Without tool support, a developer must search for contradictions in the *rule antecedent* of a rule manually, if a *rule consequent* is not executed. For this purpose, all atoms of the rule need to be individually examined by hand, which often includes inspecting each corresponding entry in the ontology. This tedious process makes testing a long and inefficient task. Therefore, there is an urgent demand for an efficient computer-aided method for debugging single SWRL Rules.

1.1 Motivation

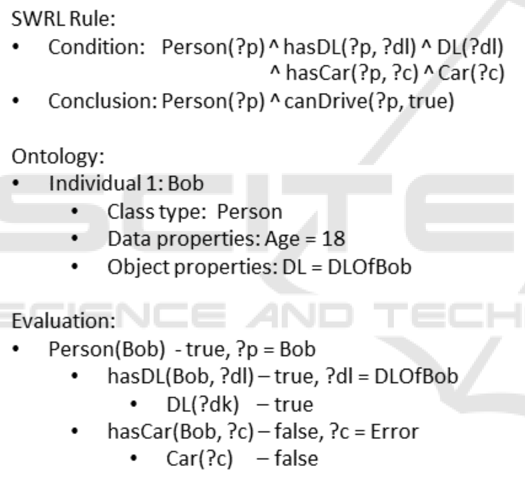
SWRL Rules consist of a rule antecedent ("if") and a rule consequent ("then"). If all conditions in the rule antecedent are satisfied, the rule consequent is considered *true* and inferences are drawn. Otherwise, the rule consequent is considered false and there is currently no method for developers to identify which of the conditions in the rule antecedent are not met and cause the rule to fail. SWRL does not provide information about the evaluation process of an executed rule. Therefore, it is difficult to identify atoms classified as false. A developer must manually resolve the rules in the logical context of the relevant entities in the ontology. In practical industrial applications (e.g. from engineering disciplines), this issue becomes considerably complicated and time consuming.

1.2 Idea

In this paper we address the task of developing a debugging method for the evaluation of single SWRL

Geyer, J., Nguyen, J., Farrenkopf, T. and Guckert, M.
Single Rule Evaluation (SRE): Computational Algorithmic Debugging for Complex SWRL Rules.
DOI: 10.5220/0006924101910198
In *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2018) - Volume 2: KEOD*, pages 191-198
ISBN: 978-989-758-330-8

rules in the context of a given ontology. The algorithm to be designed must potentially compute the output values for each atom in the rule antecedent and must be able to process rules with the help of arbitrary reasoners. Therefore this approach differs from existing open source solutions, like the *Protégé* plugin *SWRL-IQ* which is based on an XSB Prolog inference engine (Elenius, 2012). By computing the output values for each atom of the antecedent, the SRE answers the question whether the conclusion of a specific rule was drawn. Our algorithm rearranges the rule structure (atom ordering) in order to examine the integrity of references for each atom in the rule antecedent and then assigns values successively following the dependencies within the rule. is the academic standard for OWL ontology editors. It has an open architecture and can easily be extended with plugins. We present such a *Protégé* plugin which iteratively evaluates SWRL rules using our algorithm. The following example illustrates the general idea.


SWRL Rule:
- Condition: Person(?p) ^ hasDL(?p, ?dl) ^ DL(?dl) ^ hasCar(?p, ?c) ^ Car(?c)
- Conclusion: Person(?p) ^ canDrive(?p, true)

Ontology:
- Individual 1: Bob
  - Class type: Person
  - Data properties: Age = 18
  - Object properties: DL = DLOfBob

Evaluation:
- Person(Bob) - true, ?p = Bob
  - hasDL(Bob, ?dl) – true, ?dl = DLOfBob
    - DL(?dk) – true
  - hasCar(Bob, ?c) – false, ?c = Error
    - Car(?c) – false

Figure 1: Example evaluation.

1.3 Outline of the Paper

The following section provides general background knowledge for the work with OWL ontologies and in particular SWRL. Section 3 describes the concept and preconditions of our SRE algorithm. In section 4, we present a practical example of application of SRE as a proof of concept. Finally, this paper explains the approach used for the implementation and showcases the developed visualisation plugin for evaluating SWRL rules.

2 BACKGROUND

OWL is a standardised general knowledge representation language defined in the official W3C OWL guide from 2004 (McGuinness et al., 2004). Being based on description logic the language is object centred with no real support for if then rules. This can be healed by extending OWL with SWRL (Horrocks et al., 2004), which offers Horn clause like rules to express such inferences.

In addition to OWL elements that mainly provide class definitions and descriptions of their contextual interrelation i.e concepts, individuals and properties, SWRL further includes elements for functions (i.e. *built-ins*) and *restrictions* (McGuinness et al., 2004). Built-ins can be described as operators similar to functions in conventional programming languages, e.g. they can be used to express mathematical operations. Results are not passed as return values but are bound to variables. The following table summarises the most significant elements that can be used in SWRL and shows their syntactic structure.

Table 1: OWL/SWRL Elements.

OWL/SWRL Element	**Syntax**
Class	Person
Individual / Class instance	Person(bob) / Person(alice)
Data Property	canDrive(bob, true)
Object Property	isSon(bob, alice)
Built-ins	swrlb:lessThan(?*age*,18)
Restrictions	integer[$>$ 0]
Variable	?*age*

2.1 SWRL Syntax Diagram

SWRL rule consists of two sets of atoms (antecedent and consequent). An atom has a name and depending on the atom type (class, data property, object property, built-in) they can contain i-objects and d-objects. An i-object can either be an individual ID or an i-variable, which is a URI referring to an entity defined in the ontology. In contrast, d-objects are either data literals or d-variables that also refer to an entity. Fig. 2 illustrates the syntactical structure of SWRL rule in more detail.

2.2 Semantic Query-Enhanced Web Rule Language

The Semantic Query-Enhanced Web Rule Language (SQWRL) is a language based on SWRL, in which the rule consequent is replaced by so-called SQWRL selection operators. For this purpose, SQWRL provides different operators such as selection of values or a function for counting the number of entries in the result set of a query. As SQWRL is based on SWRL, it

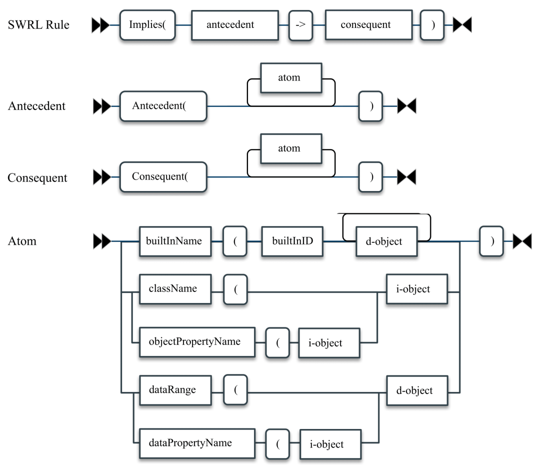


Figure 2: SWRL syntax diagram.

makes use of syntactically similar rules to specify the extraction of the data. The consequent of the rule is replaced by a retrieval specification (a set of SQWRL selection operators) in order to specify the data that should be extracted. In contrast to the retrieval specification (rule consequent), the antecedent follows the regular SWRL syntax to specify patterns for the query (O'Connor and Das, 2009). The following example illustrates a SQWRL rule which selects all individuals in the ontology that belong to concept *person* and possess a driver licence.

Person(?p) ˆ hasDL(?p, ?dl) → sqwrl:select(?p)

The operator *sqwrl:select* is one of the core functions that return a list of relevant concepts i.e. persons ?p (see (O'Connor and Das, 2009)).

2.3 Dependencies between SWRL Atoms

Typically, variables are used in more than one SWRL within a rule or a part of the rule. A fact that defines logical dependencies e.g. through object properties used. SWRL atoms that are not dependent on any other atoms are called *root atoms* (see Fig. 3). Atoms that are linked to other atoms are called *child atoms*.

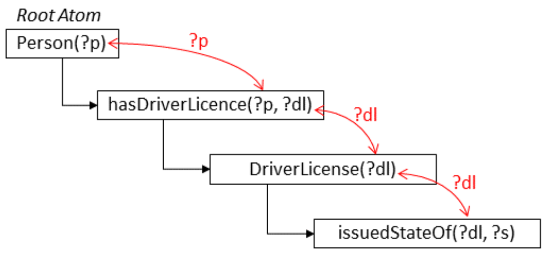


Figure 3: Dependencies between SWRL atoms.

3 SRE ALGORITHM

3.1 Specification

SRE allows debugging SWRL rules by cascading values of variables along the logical dependencies between the atoms of the rule. The debugging procedure is supposed to deliver detailed output for each atom in the rule antecedent to explain whether and why a rule consequent was executed or not. Moreover, the SRE algorithm provides an option to document results from the evaluation.

3.2 Preconditions

In order to use SRE, the SWRL rules must meet certain conditions. While the general SWRL syntax, allows rules without any class atoms in the antecedent, our algorithm depends on the convention that class atoms must explicitly be defined for each variable occurring in the rule. We can later improve the algorithm by implicitly asserting variables without binding class atom to be of type *Thing*, which we have not done yet because of performance considerations.

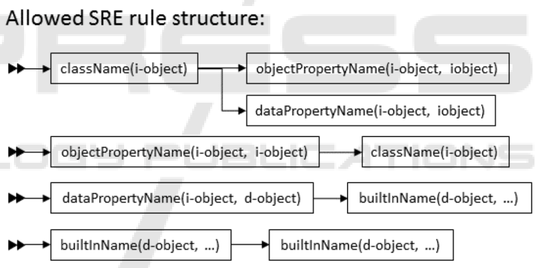

Allowed SRE rule structure:
className(i-object)
objectPropertyName(i-object, iobject)
dataPropertyName(i-object, iobject)
objectPropertyName(i-object, i-object)
className(i-object)
dataPropertyName(i-object, d-object)
builtInName(d-object, ...)
builtInName(d-object, ...)
builtInName(d-object, ...)

Figure 4: SRE Rule Structure.

As indicated in Fig. 4 SRE does not allow for object property atoms to be directly followed by another property atom. The SRE rule syntax convention requires object properties to be explicitly followed by a class atom. Otherwise, there will be errors in the SRE path creation. If this is not guaranteed, it may happen that the object property is appended by more than one atom at the same time (see Fig. 5).

3.3 Algorithm Procedure

First, the algorithm selects a rule and restructures into an internal representation. As SRE only focuses on proving whether a rule consequent has been executed, the consequent is not relevant for the evaluation. After the rule is rearranged, the algorithm detects root atoms. Variables in root atoms must be bound to user input (data literals or individual IDs) as they cannot

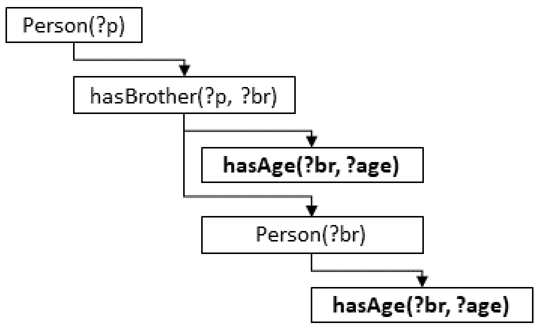


Figure 5: Inconsistent Rule Syntax.

be derived from other atoms. This input is passed to SQWRL queries which are used in the evaluation process. The algorithm creates a list in which results of the evaluation are stored, so that the user debugging the rule is able to get efficient access to them afterwards.

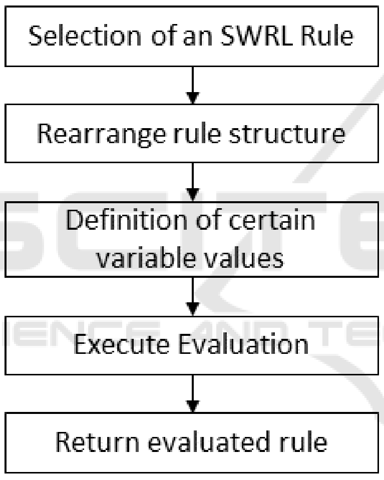


Figure 6: Algorithm procedure.

The following subsections provide a more detailed explanation of the main processes.

3.3.1 Rearrange Rule Structure

In order to execute an evaluation, the rule antecedent is rearranged into linked data structure that displays single atoms and their dependencies to each other. Therefore, the rule is split into single SWRL atoms first. After that root atoms are identified using a recursive approach that inspects each atom and its dependencies. These atoms become root nodes. Dependencies can derive from all atom types and are represented as a directed link according to the order in which variables appear as arguments. Based on this, a depth-first search is used to find all relevant child atoms. The result is a tree like structure with atoms as nodes and dependencies as edges. Note that SRE can not handle theoretically possible cyclic dependencies of variables at the moment. This tree structure called *rule graph* is then used in the evaluation process.
The following example shows how the rule antecedent is rearranged as a linked-list as illustrated in Fig. 7

Person(?p) ˆ hasDl(?p, ?dl) ˆ DriverLicense(?dl) ˆ hasAge(?p,?age) ˆ swrlb:greaterThan(?age, 18)

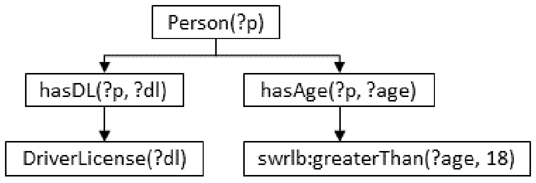


Figure 7: Rearranged Rule.

3.3.2 Definition of Variable Values

After the algorithm has rearranged the rule atoms and built the *rule graph*, variables used in root atoms of the antecedent need to be replaced by literals externally, i.e. by the user debugging the rule. In order to determine these variables, SRE first identifies all class atoms and all data property atoms in the rule list. Object properties are not relevant for this search, as the rule convention (see Section 3.2) ensures that all object property variables that can be replaced by literals, can also be found in the corresponding class atom. Moreover, it is ensured that an object property is followed by the referred class atom. Built-ins, on the other hand, only provide variables to be replaced, if they represent a function which binds a resulting value into the first argument of of the built-in atom. This is the case for example with math built-ins such as *swrlb:add*.

Fig. 8 shows the variables which can potentially be defined by a user.

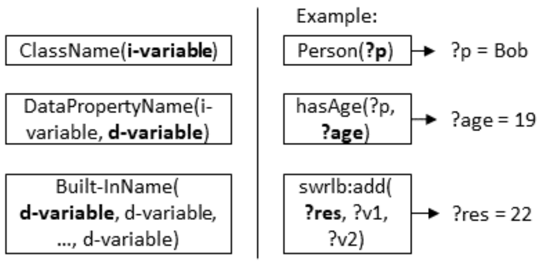

Example:
ClassName(i-variable)
Person(?p)
?p = Bob
DataPropertyName(i-variable, d-variable)
hasAge(?p, ?age)
?age = 19
Built-InName(d-variable, d-variable, ..., d-variable)
swrlb:add(?res, ?v1, ?v2)
?res = 22

Figure 8: Variable Definition.

It is essential that variables in root atoms of the rule graph are replaced by literals before the evaluation starts. This is a mandatory step as otherwise SRE cannot produce results. In this case, SRE instantly fails to evaluate the first atom, as the algorithm produces an invalid SQWRL query with an empty retrieval specification (see table 2 for valid SQWRL statements).

3.3.3 Evaluation

The evaluation step is supposed to find atoms that cause the selected rule to fail. For this purpose, the externally defined values (see chapter 3.3.1) are copied into two lists, called *master-list* and *path-list* respectively. In the beginning both lists contain the same variables and their assigned values. The two lists are crucial for evaluating built-ins used in the rule. The *path-list* contains variables together with their values that were found in a path found by the depth-first search through the rule graph. This is necessary as evaluated atoms may lead to multiple entries in the result set of the SQWRL query. Multiple results create new paths, and each paths requires its own list. The following example clarifies the issue.

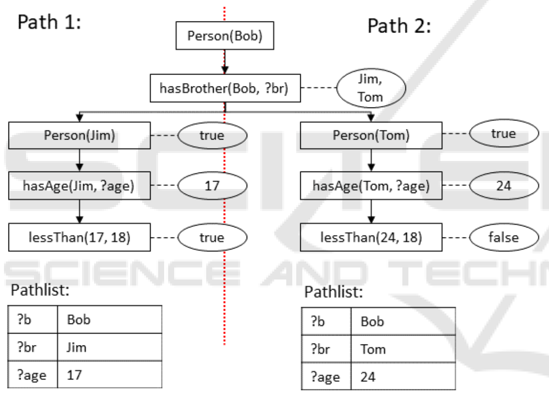


Figure 9: Pathlist.

The *master-list* is filled with variables and the corresponding values found during the evaluation process. This list is used, whenever the SWRL rule contains a built-in atom which refers to more than one other atom. In this case, values from more than one *path-list* are used. The following example illustrates the role of this *master-list*.

The evaluation uses a depth-first search to iterate through the rearranged rule list (see section 3.3.1). During each iteration, the current SWRL atom needs to be classified to find variables that can be replaced by literals. After identifying the relevant variables, the algorithm searches the *path-list* for the corresponding value. In case a matching value is found, the sqwrl:select operator is used to query the ontology. Moreover, if an atom is an instance of a built-in which refers to more than one other atom,

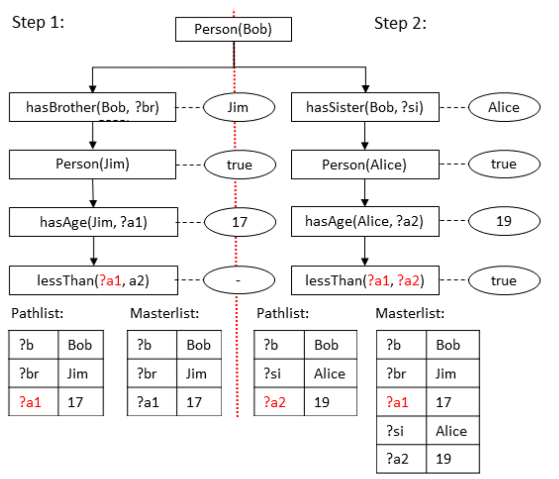


Figure 10: master-list.

the algorithm uses the *master-list* to find the required values.

For example in a query that checks whether Person(*Bob*) owns Cars(?c) the SQWRL query returns a list of all car individuals that belong to Person(*Bob*) according to object properties in the ontology.

$$ownsCar(Bob, ?c) \rightarrow sqwrl{:}select(?c)$$

The sqwrl query delivers a set of values for queried variables. These values are put into the *path-list* and the globally accessible *master-list*. The *path-list* is only valid until the depth-first search switches to another path. The following table defines the the different SQWRL queries that are required for each atom type.

Table 2: SQWRL Queries.

Atom type	**Syntax**
Class	*class(?i_obj) → sqwrl:select(true)*
Data Property	*dp(?i_obj,?d_obj) → sqwrl:select(?d_obj)*
Object Property	*op(?i_obj1,?i_obj2) → sqwrl:select(?i_obj2)*
Built-in (Typ 1)	*bi(?res, ?d_obj1, ..., ?d_objN) → sqwrl:select(?res)*
Built-in (Typ 2)	*bi(?d_obj1, ?d_obj2) → sqwrl:select(true)*

A further rule list is simultaneously created and documents the steps of the evaluation process. For each SQWRL query which examines the outcome for an atom a new entry containing information about the current SWRL atom is appended to this list. The entry is labelled as false if the query does not return any data from the ontology.

4 EXAMPLE

In this section, an example of an SRE application provides a proof of concept.

4.1 Simple Evaluation Example

This example shows a simple evaluation process of SRE. The following entries are provided by the given ontology.

- **Classes:** Person, Dl(short form for "Driver Licence")
- **Instances:** Bob instance of Person, DL_Of_Bob instance of Dl
- **Relations:** Bob hasDriverLicence DL_Of_Bob, Bob hasAge 17

Based on this, the following example rule is to be evaluated:

Person(?p) ^ hasDriverLicence(?p, ?dl) ^ Dl(?dl) ^ hasAge(?p, ?age) ^ swrlb:greaterThan(?age, 18) → Person(?p) ^ canDrive(?p, true)

At first, the rule is rearranged into the structure of the corresponding *rule graph*. As the rule consequent is not relevant for the assessment, it can be discarded. The rule antecedent is now arranged as as shown in Fig. 11.

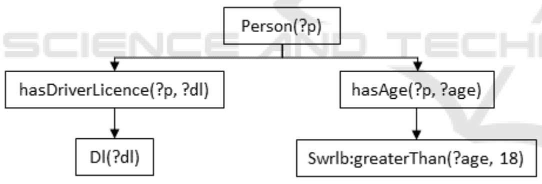


Figure 11: Example 1: Rearranged Rule.

In the next step the *rule graph* is used to identify variables that can be replaced by literals. Root atoms of the *rule graph* must be replaced, as they define the starting point for the evaluation. In our example variable ?p is identified as root atom and is replaced by the literal "Bob". The variable is now used in the corresponding SQWRL query.

As Fig. 12 shows person Bob is not allowed to drive a car, because the built-in *swrlb:greaterThan(?age, 18)* returns *false*.

5 IMPLEMENTATION

Based on the concept of SRE we implemented the algorithm as a Java-based API. Java was chosen for compatibility as the programming interfaces and the ontology editor *Protégé* are also written in Java. The following sections describe the basis architecture of the SRE API and how it is used to implement the prototypical SRE *Protégé* Viewer.

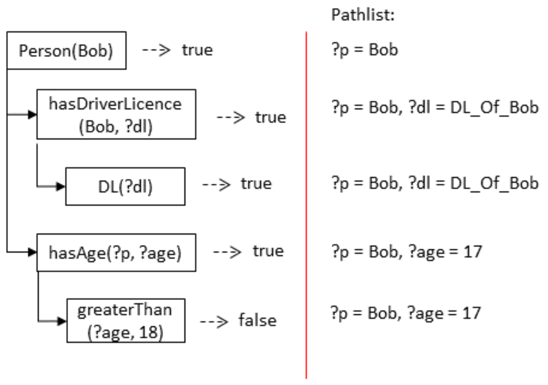

Pathlist:
Person(Bob) --> true
?p = Bob
hasDriverLicence (Bob, ?dl) --> true
?p = Bob, ?dl = DL_Of_Bob
DL(?dl) --> true
?p = Bob, ?dl = DL_Of_Bob
hasAge(?p, ?age) --> true
?p = Bob, ?age = 17
greaterThan (?age, 18) --> false
?p = Bob, ?age = 17

Figure 12: Example 1: Evaluation Process.

5.1 Single Rule Evaluation API

In order to make the SRE reusable the algorithm was implemented in the form of an application programming interface (API). This API focuses on usability and provides different interfaces to monitor the progress of the evaluation. The implementation follows architectural principles i.e. it conforms to design patterns especially the observer and the facade pattern. The facade pattern provides a single entry point for users which summarises the most important methods. This entry point (facade class) provides functions such as the registration of SRE observers and SRE main clients. Those two interfaces are used for the observer pattern. A user that is registered as a `SREMainClient` is able to define literals required to be assigned to variables during rule evaluation. In contrast, *SREObserver* instances only receive log messages and the evaluated rule after the assessment has finished. The evaluation can be started, only when the user has registered all observers. Moreover, by using the *executeEvaluation()* function, a rule is selected by addressing its rule name. The facade class provides a function for this specific purpose.

When an ontology is submitted, the *ExecutionUnit* initialises the *OntologyAccess* component, which provides a utility class to enable bi-directional data exchange between the ontology and the system. This utility class is mainly used by the *EvaluationEngine*, which is responsible for the rearrangement process, the identification of to be replaced variables, and the evaluation of a selected rule. Queries against the Ontology are generated by the *QueryGenerator* component and run by the *EvaluationEngine*. It provides an interface that returns SQWRL queries based on data delivered by the *EvaluationEngine*. The *NodeGenerator* component documents the progress of the evaluation. It builds a *rule graph*, which is similar to the *rule graph* for a rearranged rule. However, it contains extra information for each evaluated SWRL atom. Because of the observer pattern, the *EvaluationEngine* is connected to an interface of the *UpdaterComponent*. This component has the purpose to handle communications with the user.

See Fig. 13 for an illustrated explanation of the API.

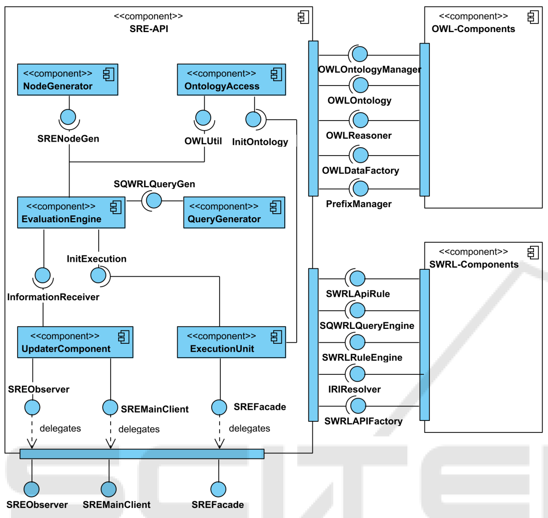
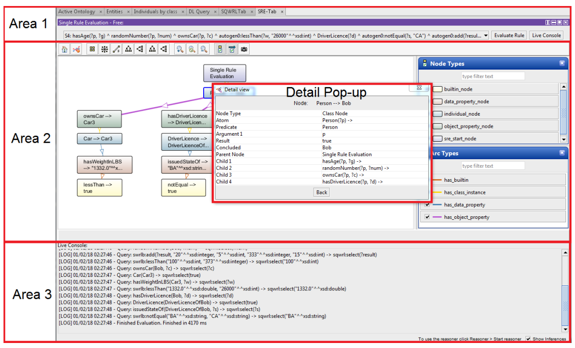


Figure 13: SRE API Component Diagram.

5.2 SRE Protégé Viewer

The implementation of the SRE Viewer as a *Protégé* plugin makes use of the developed SRE API. The plugin provides a graphical user interface via the *Protégé* ontology editor and is divided into three subpanels. The first area contains a drop-down menu in which a user can select the rule for the evaluation. In order to execute the evaluation with a selected rule, the panel provides a button that starts the process. The panel in the middle of the perspective contains a canvas area on which the results of an evaluated rule are displayed. As the assessed rule is a linked list of nodes, it is displayed as a tree structure. For this, the canvas area is inspired by the *Protégé* plugin *Ontograf*, which is based on the Cajun Visualization Library (Falconer, 2010). Finally the GUI provides a simple output console that displays log messages that the SRE API produces while the evaluation is running. A single click on a node shows detailed information about the assessment of a specific SWRL atom in a pop-up window.



S4: hasAge(?p, ?g) ^ randomNumber(?p, ?num) ^ ownsCar(?p, ?c) ^ autogen0:lessThan(?w, "26000"^^xsd:int) ^ DriverLicence(?d) ^ autogen0:notEqual(?s, "CA") ^ autogen0:add(?resul...

Node: Person --> Bob

Node Type	Class Node
Atom	Person(?p) ->
Predicate	Person
Argument 1	p
Result	true
Concluded	Bob
Parent Node	Single Rule Evaluation
Child 1	hasAge(?p, ?g) ->
Child 2	randomNumber(?p, ?num) ->
Child 3	ownsCar(?p, ?c) ->
Child 4	hasDriverLicence(?p, ?d) ->

Live Console:
```
[LOG] 01/02/18 02:27:46 - Query: swrlb:add(?result, "20"^^xsd:integer, "5"^^xsd:int, "333"^^xsd:integer, "15"^^xsd:int) -> sqwrl:select(?result)
[LOG] 01/02/18 02:27:46 - Query: swrlb:lessThan("100"^^xsd:int, "373"^^xsd:integer) -> sqwrl:select("100"^^xsd:int)
[LOG] 01/02/18 02:27:46 - Query: ownsCar(Bob, ?c) -> sqwrl:select(?c)
[LOG] 01/02/18 02:27:47 - Query: Car(Car3) -> sqwrl:select(true)
[LOG] 01/02/18 02:27:47 - Query: hasWeightInLBS(Car3, ?w) -> sqwrl:select(?w)
[LOG] 01/02/18 02:27:47 - Query: swrlb:lessThan("1332.0"^^xsd:double, "26000"^^xsd:int) -> sqwrl:select("1332.0"^^xsd:double)
[LOG] 01/02/18 02:27:48 - Query: hasDriverLicence(Bob, ?d) -> sqwrl:select(?d)
[LOG] 01/02/18 02:27:48 - Query: DriverLicence(DriverLicenceOfBob) -> sqwrl:select(true)
[LOG] 01/02/18 02:27:48 - Query: issuedStateOf(DriverLicenceOfBob, ?s) -> sqwrl:select(?s)
[LOG] 01/02/18 02:27:48 - Query: swrlb:notEqual("BA"^^xsd:string, "CA"^^xsd:string) -> sqwrl:select("BA"^^xsd:string)
[LOG] 01/02/18 02:27:48 - Finished Evaluation. Finished in 4170 ms
```

Figure 14: SRE Protege Viewer.

6 CONCLUSION AND FUTURE WORK

In this paper we introduce a new method for evaluating SWRL atoms in the rule antecedent of SWRL rules in order to examine whether a rule consequent will be executed. This can significantly simplify the work on complex ontologies as it reduces the effort to find atoms that cause rules to fail. The SRE algorithm was implemented as an application programming interface (API) that focuses on usability and creates new possibilities for other projects as it can easily be integrated in application programs. Moreover the SRE and the developed API offer various applications for real world problems, like form-input validation with the help of SWRL rules. We also implemented a prototypical SRE Viewer as a plugin for Protégé since we believe that it can become a useful tool for future applications. The visualisation of the evaluated rule simplifies the debugging process, as the atoms that produce *false* values are instantly visible. The developed plugin at this stage offers potential for extensions and improvements. A first problem to be solved is caused by the SQWRL query language as large ontologies may cause heap space errors which are due to the underlying reasoning unit. Moreover, the development of an SRE syntax validator is a useful improvement, as it supports the editing of complex SQWRL rules. This validator can inspect rules suggest changes, if they do not conform to defined conventions.

We will provide the source code for the SRE project on github. The repository can be accessed under the following URL: https://github.com/KITE-Cloud/SRE

REFERENCES

Fluent editor 2015 help.

Elenius, D. (2012). Swrl-iq: A prolog-based query tool for owl and swrl. In *OWLED*.

Falconer, S. (2010). Ontograf. *Protégé Wiki*.

Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., Dean, M., et al. (2004). Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21:79.

MacLarty, I., Langevine, L., Bossche, M. V., and Ross, P. (2016). Using swrl for rule-driven applications.

McGuinness, D. L., Van Harmelen, F., et al. (2004). Owl web ontology language overview. *W3C recommendation*, 10(10):2004.

O'Connor, M. (2018). How to debug swrl rules?. `https://mailman.stanford.edu/pipermail/protege-owl/2008-May/007034.html`. Last checked on April 23, 2018.

O'Connor, M. and Das, A. (2009). Sqwrl: a query language for owl. In *Proceedings of the 6th International Conference on OWL: Experiences and Directions-Volume 529*, pages 208–215. CEUR-WS. org.