Explainability in Expert Systems

Angelo Kastroulis
akastroulis@carrera.io
Carrera Group

ABSTRACT

It has been said that expert systems (rules engines and code-based execution engines) are inherently explainable. But how, exactly? This paper explores the topic and surveys the landscape of methods of "explainability", or "interpretability".

ACM Reference Format:
Angelo Kastroulis. 2022. Explainability in Expert Systems. In *Proceedings of* , , , 3 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Expert systems such as rules engines attempt to separate the authoring of rules from their execution [3]. A computer scientist who would "hard code" a rule would naturally logically compose, segregate, and optimally code the rules such that they would be easy to understand and execute quickly. However, computer scientists are not always the best resource to create the rule because they may not be the domain expert. An expert system seeks to close the gap by allowing rules to be written in a domain-specific language, separating the knowledge required to author it from the expertise required to execute it. The execution engine, therefore, must perform optimization techniques to reorganize the rules and execute them in the appropriate manner [1].

There are two popular methods for execution of rules: forward-chaining and backward-chaining [1].

1.1 Forward Chaining

A forward-chaining rules engine is one that is "data-driven" [1]. It asserts data into the system and executes one or more rules on the data, propagating the facts through the rule chain.

1.2 Backward Chaining

A backward-chaining rules engine is "goal-driven" [1]. It starts with the conclusion which the engine tries to satisfy. If it cannot satisfy a conclusion, it searches of others that it can that perhaps may make their way to the current goal. Prolog is an example of backward-chaining, although Drools can also be configured to work in that way [1].

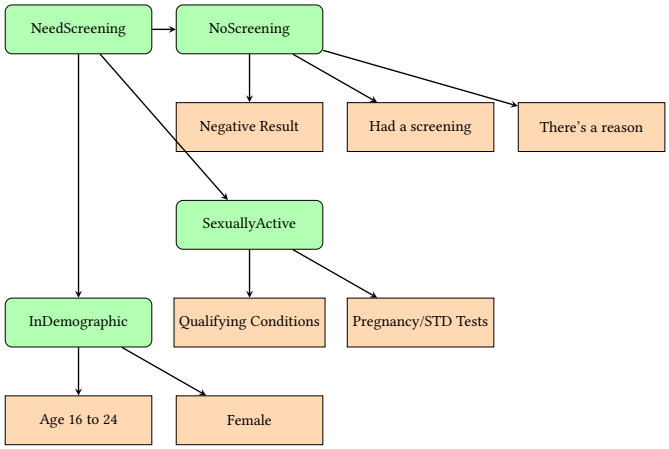

Figure 1: Rules from Listing 1

1.3 Explainability

Since the rules are written by an expert, presumably, the expert is able to understand and interpret them. However, it's not an automatic process. This paper describes common approaches used to explain rules.

2 BACKGROUND AND RELATED WORK

Rules can be complex. Consider the rule as described in the rule in Listing 1 (written in Clinical Quality Language). The "top level" rule is "NeedScreening". Need screening, requires rules "InDemographic", "SexuallyActive", and "NoScreening", which could (but do not in this example) require execution of other rules.

Figure 1 shows the rules in terms of their call stack. The rules (green boxes), in term, execute some logic (the orange boxes). Each rule will return data to the previous rule for execution. So, one method of explanation could consist of the call stack and the corresponding data at each step. For example, at the bottom of the stack, SexuallyActive executes on all of the patient's conditions (pregnancy, STDs, reproductive conditions), returning a matching subset along with diagnostic orders (pregnancy and STD tests) which also return a subset. Sexually active, in turn, passes a value up the chain to NeedScreening, which received data from all of the other rules that fired.

The amount of data, then, is equivalent to sum of the data required for every rule, and may be duplicated many times (for example, if different rules checked conditions).

Listing 1: Domain Specific Language Rule Example (CQL)

```
library ChlamydiaScreening_CDS version '2'

using QUICK

codesystem "SNOMED": 'http://snomed.info/sct'

valueset "Female Administrative Sex": '...'
valueset "Other Female Reproductive Conditions": '...'
```

```
valueset "Genital Herpes": '...'
valueset "Genococcal Infections and Venereal Diseases": '...'
valueset "Inflammatory Diseases of Female Reproductive Organs": '...'
valueset "Chlamydia": '...'
valueset "HIV": '...'
valueset "Syphilis": '...'
valueset "Complications of Pregnancy": '...'
valueset "Pregnancy Test": '...'
valueset "Pap Test": '...'
valueset "Lab Tests During Pregnancy": '...'
valueset "Lab Tests for Sexually Transmitted Infections": '...'
valueset "Chlamydia Screening": '...'

context Patient

define "InDemographic":
        AgeInYears() >= 16 and AgeInYears() < 24 and "Patient"."gender"
        in "Female Administrative Sex"

define "SexuallyActive":
        exists (["Condition": "Other Female Reproductive Conditions"])
        or exists (["Condition": "Genital Herpes"])
        or exists (["Condition": "Genococcal Infections and Venereal Diseases"])
        or exists (["Condition": "Inflammatory Diseases of Female Reproductive Organs"])
        or exists (["Condition": "Chlamydia"])
        or exists (["Condition": "HIV"])
        or exists (["Condition": "Syphilis"])
        or exists (["Condition": "Complications of Pregnancy"])
        or exists (["DiagnosticOrder": "Pregnancy Test"])
        or exists (["DiagnosticOrder": "Pap Test"])
        or exists (["DiagnosticOrder": "Lab Tests During Pregnancy"])
        or exists (["DiagnosticOrder": "Lab Tests for Sexually Transmitted Infections"])

define "NoScreening":
  not exists (["DiagnosticReport": "Chlamydia Screening"] R where R."issued" during
    Interval[Today() - 1 years, Today()] and R."result" is not null)
  and not exists (["ProcedureRequest": "Chlamydia Screening"] P
    where P."orderedOn" same day or after Today())
  and not exists (["Observation": "Reason for not performing Chlamydia Screening"])

define "NeedScreening": "InDemographic" and "SexuallyActive" and "NoScreening"

//The following previously read "ProcedureRequest" where it currently reads "Tuple"

define "ChlamydiaScreeningRequest": Tuple {
  type: Code '442487003' from "SNOMED"
  display 'Screening for Chlamydia trachomatis (procedure)',
  status: 'proposed'
  // values for other elements of the request...
}
```

3 APPROACHES

It is easy to see that there will be volumes of data generated, depending on how large the working memory set is. A decision graph may contain hundreds of nodes and thousands of points of data.

Approaches can be grouped into 2 categories: author-driven, or engine-driven. Author-driven approaches encode knowledge into the process and is more easily able to eliminate noise (since the author knows which data is critical) at the cost of a more laborious authoring process.

An engine-driven approach will produce the most data (and noise), requiring downstream systems (and users) to sift through the information to determine which component they desire to have more information on.

3.1 The Author-driven Method

While the rule in Figure 1 was very easy to follow, consider the rule in Figure 2. It is difficult for the end user to determine what the meanings of MostRecentFeature, MostRecentAsserted, InferenceMostRecentAsserted and CommonAssertedObservation mean. Those have meaning to the rule author, but to the end user who may want more information as to why a decision was made, it is not immediately obvious. In those cases, an approach that allows the author to encode their thought process and return a friendly

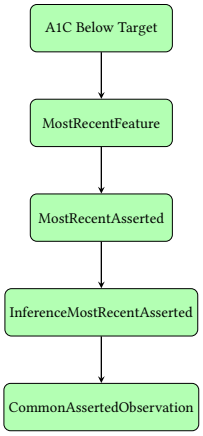

Figure 2: Example of a rule that is difficult to follow.

message is more desirable. Listing 2 is an example of a possible approach coded for a Drools-based rules engine.

Listing 2: Drools example with explanation

```
import com.sample.TestExample;
import com.sample.TestExample.Message;

rule "Hello World"
when
    m : Message( product == Message.tv, myMessage : message )
then
    m.setMessageText( "The product is a TV" );
end

rule "GoodBye"
when
    n  : Message( product == Message.smart, myMessage : message )
then
    m.setMessageText( "The product is a Smart" );
end

rule "Else condition"
when
    e  : Message(product != TestExample.productClass)
then
    m.setMessageText( "The product is a " + TestExample.productClass );

end
```

3.2 The Call Stack Method

One method that does not require information from the author of the rule (but results in the largest amount of data), is to capture the entire call stack and the data at each step. Given the size of the data, it makes returning the explanation along with the rule impractical. Listing 3 shows an example log of the call stack, which illustrates the difficulty in understanding how to interpret the execution stack (especially when it contains many thousands of lines.

In Listing 3, the arrow next to the line number ("->" and "<-") denote the direction (enter and exit, respectively) of the call. The function is entered in line 1, traverses through hundreds of operations, then unwinds and exits on line 929.

Listing 3: Example call stack log

```
Evaluating 'MyFunction' in 'MyLibrary' for 'data.json'...
Line 001->[1] [FunCall#1] input = {...}
...
Line 530<-[16] [FunCall#1>NotEmpty$#4>Let($filter$source)#5>Conditional#20>Filter#26>
    Let(C)#29>Let($and$left)#31>Conditional#160>Let($and$right)#164>In$#165>
        Let($mappend$source)#166>Let($filter$source)#167>Let($mappend$source)#168>
            Conditional#218>NotEquals$#219>Not$#222] ENV: {some environment}
...
Line 926<-[4] [FunCall#1>NotEmpty$#4>Let($filter$source)#5>Conditional#20] result =
```

```
    Col(LazyList(Datum({some data}))))
Line 927<-[3] [FunCall#1>NotEmpty$#4>Let($filter$source)#5]
    result = Col(LazyList(Datum({some data}))))
Line 928<-[2] [FunCall#1>NotEmpty$#4] result = Bool(true)
Line 929<-[1] [FunCall#1] result = Bool(true)
```

3.3 Listeners and Exporting Data

Another method is to attach a "listener" to each of the rules [2]. The listener, then sends the rule and all of its corresponding data to an analytic database that can be queried for the important information. Listing 4 is an example of a Drools listener that will intercept the agenda and can then send the data to systems (like an analytics engine). Listing 5 is an example of intercepting the entire working memory. The advantage of the listener approach is that the system allows configurability as to the types of events that should be streamed, thus reducing noise.

Listing 4: Example code monitor for JBoss Process Manager agenda [2]

```
ksession.addEventListener( new DefaultAgendaEventListener() {
   public void afterMatchFired(AfterMatchFiredEvent event) {
       super.afterMatchFired( event );
       System.out.println( event );
   }
});
```

Listing 5: Example event listener for all of working memory [2]

```
ksession.addEventListener( new DebugRuleRuntimeEventListener() );
```

4 CONCLUSION

There are several methods of implementing explainability in rules engines. The two types of approaches: author-driven and engine-driven. Author-driven approaches have the advantage of encoding friendly messages and decoupling the intent from the actual rule code.

Engine-driven approaches are generic and require no special effort per rule, but trade that for cryptic and voluminous data.

While each method has its merits, the goal of the explanation makes an impact. For example, in cases such as forensic analysis or debugging, the entire call stack, or listener may be more valuable as it provides the most information. However, in cases such as end-user feedback (for the purposes of the user knowing which corrective action to take), a more user-curated message is necessary, and therefore an author-driven approach is appropriate.

The listener approach sits somewhere in the middle and is desirable if a heuristic can be developed alongside the content to describe which events are interesting. Unfortunately, not all domain-specific languages offer that approach (for example, Clinical Quality Language does not allow annotations of rules, and Drools requires that listeners be coded in Java). In those cases, adding a listener to all events is, in effect, the same as the call-stack.

REFERENCES

[1] 2011. JBoss Drools Documentation Chapter 1. (2011). https://docs.jboss.org/drools/release/5.3.0.Final/drools-expert-docs/html/ch01.html

[2] 2021. Chapter 8. Decision Engine Event Listeners and Debug Logging Red Hat Process Automation Manager 7.5. (2021). https://access.redhat.com/documentation/en-us/red_hat_process_automation_manager/7.5/html/decision_engine_in_red_hat_process_automation_manager/engine-event-listeners-con_decision-engine

[3] Suresh Subramoniam. 1992. Expert Systems: Guidelines for Managers. *Industrial Management & Data Systems* (1992).