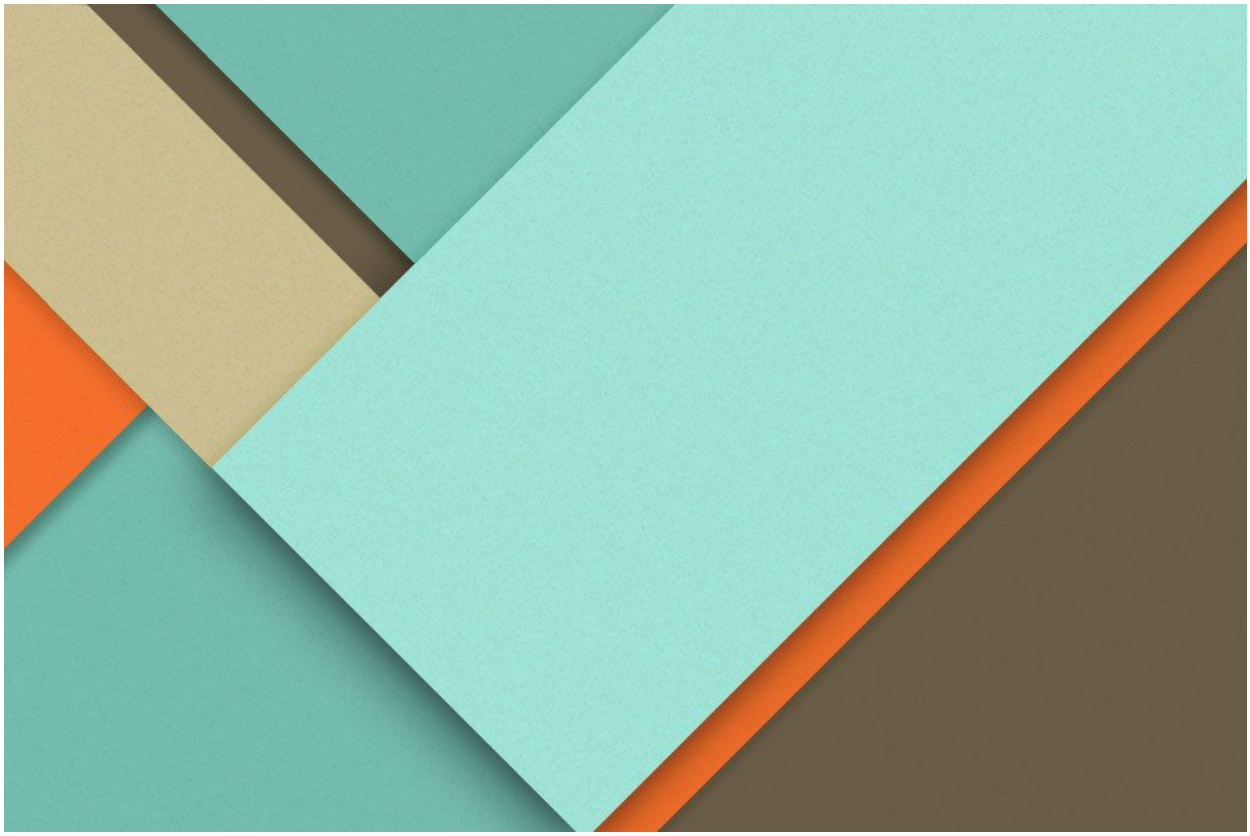


Simple Rule Based Inference Engine Final

03.16.2017

Trevor Ching, Juan Huerta, Eliezer Miron, Kelvin Silva
CMPS 109, Dr. Karim Sobh
Computer Science Department
University of California, Santa Cruz

Overview

A Simple Rule Based inference engine (SRI) is a program which stores facts with parameters and applies logical rules to filter out, and perform variable substitutions within the parameters to return results from an inference query. SRI is mainly a command line terminal program whereby the user types in commands to add or remove facts and rules, and also inference queries. Lastly, the engine must be able to load an SRI File and write to an SRI File on disk which stores facts, rules, and inference queries for later use.

Our team used C++ and STL Library as implementation. We used functionalites up to the C++ 14 standard.

To compile, type make. G++ will be invoked with --std=c++0x. The output executable will be called sriProject.

Assumptions

1. We assume that the number of parameters in a fact or rule is capped at 2.
2. We assume that the number of inputs for the AND and OR operators is capped at 2.
3. We are (temporarily) assuming that no two rules share the same name.
4. We assume the input for AND rules follows that of the example in the pdf given.
5. Dump on client, load on client. Send rules and commands over the wire in buffer strings.
6. Server and Client connect on 0.0.0.0 port 2222
7. SRIServer has a read buffer of 8192 bytes. SRIClient has a read buffer of 65535 bytes
8. TCPSocket, TCPServerSocket made by Prof. Karim. Connection class adapted from lectures

Specifications, and Class Design Summary

Overview Of Design

The project is split into two components, a server and client.

The **SRIServer** is the component which contains KnowledgeBase, RuleBase, QueryOperations, and all classes from previous assignments. The only difference is that we added a layer ontop of the interface (Interface Class), which was the TCPServerSocket and Connection classes and routines to receive commands from a simple client. The servers are sent via plaintext from the client, and parsed and processed by the server. Upon completion of a query and compilation of results, the server sends back in plaintext the results to the client for printout to the user.

The SRIServer socket layer is multithreaded and allows for connection and service to multiple clients.

The **SRIClient** is just a simple program which retrieves a line from the user and sends it over the wire to the sri server. It does not do any processing of queries, or any lookup, or parsing. The only parsing that the client does is to check for EXIT, LOAD, or DUMP processing. Heavy operations are offloaded to the server, since the client is supposed to be a lightweight program designed to interface with the server. The SRIClient uses the TCPSocket class as an abstraction to unix sockets for communication to the SRIServer.

SRIClient Specifications

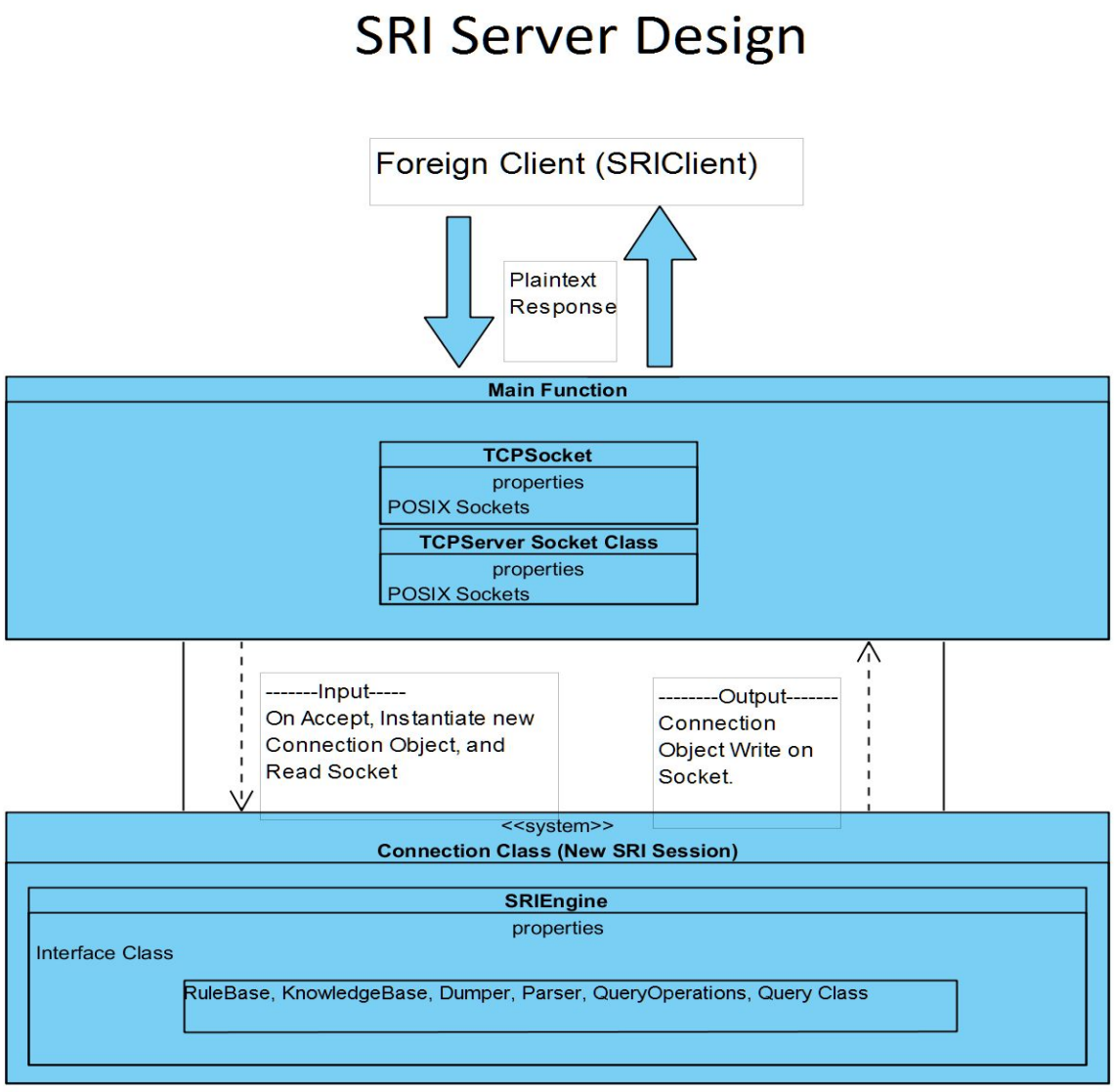
The SRI Client is a simple application which instantiates a socket on 0.0.0.0:2222 and receives input from the user, writes to the socket, and reads a reply from the SRI Server.

It uses the TCPSocket class (described in SRI Frontend). It can load an SRI file on hard disk and send it to the SRI Server. SRI Client can also dump, which means that the SRI Engine, on a dump operation, will synthesize all information in Knowledge and Rule base into a string and send it back to the client so that the client can write it to a file.

SRIServer Specifications

The SRI Server is another application built in two parts: the main function TCPServer Engine and Connection Class (Front End), and the SRI Engine (Back End) which contains Knowledge Base, Rule Base, Parser, Interface, Dumper, Query Operations. The backend loader is unused as this functionality has been implemented in the front end, since all that it requires is to create plaintext, or receive plaintext and successively make queries to the interface. Dumper collects a string version of an SRI database on the server-side, and sends that information to the client.

SRI Server Design (Front End):



Prof. Karim. TCPSocket Class

The TCPSocket class encapsulates all information necessary to instantiate a POSIX socket, and perform read and write, shutdown operations. (Was made by Prof. Karim)

Prof. Karim. TCPServer Socket Class

Encapsulates TCPSocket class to receive connections (Blocking Accept).

Connection Class

Connection class is a wrapper that encapsulates the SRI Engine (Backend), and Thread classes. For each accept on the listening socket of the server, the accepting socket (for the new client) is passed to the connection object, and a new connection object is instantiated (along with a new SRI Engine backend). The new connection object is then ran on a new thread which runs a routing of blocking reads and writes to interact with its client. Since it is ran on a new thread, the server can continue execution by listening to accept new clients.

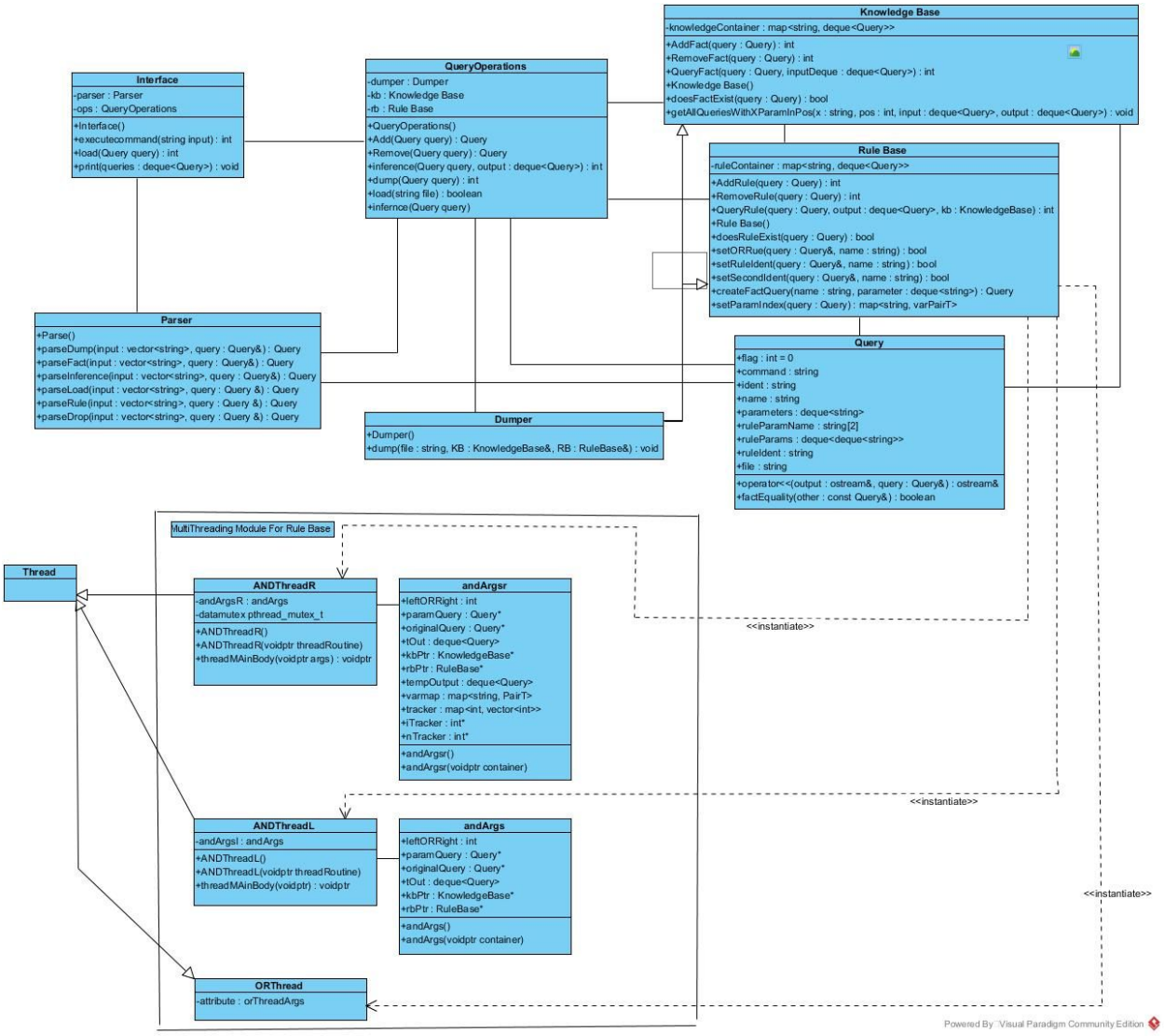
Connection object will retrieve plaintext from the client's socket and pass it to the sri engine. Once the sri engine returns to the connection object with a plaintext reply, the connection object will write to the socket and send to the client.

Connection class is regarded as the middle end.

SRI Server SRI Engine Design (Back End) (Class Diagram):

Shown Below is the SRI Server Engine (Back End) design. The interface class is through which the user (in this case, the user is the front end socket class) can convert plaintext into query objects and receive plaintext replies from the engine. The socket classes interact with the interface by inputting plaintext and receiving plaintext. The role of the SRI Engine Backend is to convert the plaintext into objects and computer logic, execute the logic, and reply back the results in plaintext, after which the Socket, TCPSocket, and Connection Class (each connection class has an interface, since multiple

connection objects may be instantiated to support multiple clients), work in tandem to send the plaintext over the wire, back to the client.



Execution of the backend begins by instantiating an Interface object which is the main driver for parsing user commands, creating query objects from parsed data, and calling the appropriate functions from Dumper, Knowledge Base, or Rule Base.

After instantiation of Interface object, we loop infinitely until the client sends in a sentinel for termination (in the Connection class). During the loop we extract input (read from socket, interact with frontend) and call Interface::executeCommand. This function parses the input, creates a query object (struct), and passes it to the appropriate query handling function in QueryOperations. The appropriate function in QueryOperations calls the proper Dumper, Rule Base, or Knowledge Base functions to Add, Remove, or Inference a fact and return to the user.

The exception to this is when Loading SRI files, in which case the Interface just executes commands, line by line, from a SRI file, where the .sri file is stored as a binary file with ascii characters and commands that are the same as what the user would type on the keyboard. In our server-client interaction, the SRI File is sent from the client to the front end, and is passed to the interface.

Rule Base Class Design

The rule base stores rules as a map with the name of the rule as the key and deque of Query objects. A query object is used for the rule base because it contains all relevant information needed for a rule already. If we did not use a query object in the map we would have to extract and insert the data into the map which requires extra unnecessary steps.

The rule base has 3 main functions, Add, Remove, Inference. The rest of the functions are helper functions for Inference.

For multithreaded functionality, OR operations are ran in parallel, and AND operations are pipelined, with multiple threads spawned for every query in the pipeline.

To implement multithreading we created 3 classes which inherit from the Prof. Karim's thread class. Each of those 3 classes overload the pure virtual method "threadMainBody" which contains the salient code which is to be ran by a separate thread.

The three classes are:

- AndThreadL --Extracts Queries from Knowledge Base from left operand in AND clause, and add to a deque
- AndThreadR --As AndThreadL extracts queries, check deque, and consume queries, query those queries to knowledge base and extract matching queries. Add to a final return deque.
- OrThread --Perform querying on both sides of OR clause in parallel to knowledge base.

Knowledge Base Class Design

The Knowledge Base class contains all the facts defined by the user, such as "Father(Bob, John)". This class can be queried with simple inferences such as "Father(Bob, $X)" or "Father($X, $Y)", or any other type of variable substitution. It is the underlying backend for the rule base class, whose job is to process complicated AND, and OR queries, and successively query the Knowledge Base Class for simple queries and filter out results which match.

The Knowledge Base class has 4 main functions (Constructors and Destructors were defaulted):

- int AddFact(Query query)
- int RemoveFact(Query query)
- int QueryFact(Query query, deque<Query>& inputDeque)
- bool doesFactExist(Query query)

AddFact takes in a query struct, and extracts the name and parameters fields, and adds it to the Knowledge Base container.

RemoveFact takes in a query struct, and extracts the name and parameter fields. It then calls QueryFact to obtain a list of all queries matching the remove command from the user. Then it removes all facts from the knowledge base container which matches results from QueryFact. This is to make sure that the user can delete many sets of rules at a time.

QueryFact takes in a query struct and a reference to a deque on which to return the outputs. It first obtains the fact under which the query specifies (For example, father, or mother). After obtaining all the facts under that name, we then do a pattern match and filter each fact which does not match the pattern. There may be various patterns, for example, under father: (Bob, John), (Bob, $X), ($X, $Y), ($X, $X). After matching all patterns and filtering out the results, they are added to the inputDeque return value parameter. (We return by reference, but specifying it as the second parameter in QueryFact)

Query Class (Struct) Design

We decided to use a struct at the beginning since we had no private members and no member functions. However as our implementation evolved, the Query Class transformed into more of a class instead of a struct. But due to the flexibility of C++ a struct and class are the same thing, and we also have no private members, so we can use the default functionality and kept it declared as a struct.

The main functionality of this class is to be passed around from Interface, to Rule Base, Knowledge Base, Dumper, and Loader objects as a message. Each class then looks inside the Query object and reads the structure fields that it needs for successful operation. Query objects were passed by value in case a certain class wanted to store it in its internal data structure. Since we had to meet a deadline, we did not want to play around with references, and speed optimizations. We focused on correct output, and functionality first, and optimizations are to be made later in the course of this project.

The query object contains the following fields:

- Int flag: Signals the presence of variables in parameter list
- String command: Name of the command issued, FACT, RULE ,INFERENCE

- String ident: Rule or fact identifier
- String name: Name of the rule or fact to add like Parent, Father
- deque<string> parameters: The parameters of a fact or rule
- String ruleParamName: name of the rule
- String deque<deque<string> > ruleParams: rule parameters
- String ruleIdent: rule identifier (AND, or OR)

Query objects may be passed only 2 or 3 times in the course of executing a user's command, thus there is not much overhead in passing by value. We make sure to only use the fields that we need for a certain operation. As such, unneeded fields are default constructed which do not take up much space in memory.

In addition, we implemented an insertion operator for the query object so that we could output to the user results of an INFERENCE. This was done for simplicity.

Lastly, the query object has a member function to test for fact equality. It compares the name and parameters of one query object to another.

Dumper Class Design

The Dumper is a class of its own, with a single central function called "Dump". It is passed a reference to a buffer string, a reference to the KB, and a reference to the RB. The Dumper class is a friend class to both the RB and KB, which allows it to directly access the rules and facts stored in three data structures. It iterates through each rule and fact, parsing them into a buffer and formatting them as necessary. This buffer is then given to the client to write to a file of the proper pathname. If the file already exists, it is overwritten. There is not really any necessity for this function to be in its own class, rather than in the QueryOperations class. This is simply vestigial design from the first iteration of class design. it is not not high priority to change it.

Load Function Design

Load function is simply a function inside of the main function. It takes a filename by taking everything after the LOAD command and loads the file locally. Then it will input each command into the socket executing the commands line by line as if the user input the commands themselves.

Interface Class and QueryOperations Class Design

The interface class is what controls the functionality of the program. It takes the parsed input and checks to see if it was given a valid command identifier. If it does, then it calls the functions from the QueryOperations class to execute.

Prof Karim's Thread Class Design

The thread class that was used was the one that was provided by Professor Sobh. It provides the functionality of checking that a thread is running, starts the execution of a thread, and blocks threads from executing until another thread finishes. This class was extended to classes used by Rule Base to be used to implement multithreading in our program. With multi-threading we provide parallelization to the OR operator and pipelining of the AND operator.

Milestones

I. Workflow Description

To successfully complete the project, our team decided to use slack messaging to set up meetings and talk about the project online. Since the university git account was closed to other members, we decided to create our own github repository to commit and save changes to avoid losing code. We also used cloud 9, an online IDE which runs a virtual ubuntu distribution. We used this because some members did not have linux installed, as well as that this software allowed us to collaborate online and has tools pre installed.

II. Class Design Changes from Last Assignment

In the first round of design, we thought about implementing the Knowledge Base as a set of nodes and edges to traverse and build up a result from an inference query. After much thought, we decided to change the Knowledge Base to utilize a dictionary to hold strings of names of facts and parameters which was much easier.

In addition, we did not have a good way of passing messages between classes in the first round of design. In this assignment we invented the QueryOperations and Query structure as a way to pass information to Rule Base and Knowledge Base, as well as Dumper and other classes.

Lastly, we decided to have the Knowledge Base directly hold query objects in its map because every query object already has names, and parameters inside of it. So it was unnecessary to extract fields from query objects and put them in a map. Holding the query objects directly in the Knowledge Base makes implementation faster and easier so that we could successfully meet project deadline.

III. Knowledge Base Testing and Implementation

The Knowledge Base's primary data structure is the map structure in C++ STL Library. We also used the Deque data structure which provides array-like access and queue and stack like functionality. Here is how all facts are held:

map<string, deque<Query> > knowledgeContainer

The first string in the map holds the name of the Fact. In "Father(Bob, John)", the father would be held as the key to the value of deque<Query>. The deque of queries holds the query object which contains the parameters for the fact "(Bob, John)". This way, the program can quickly filter out facts with different names, ex. "Apple(X, Y)" and "Father(X, Y)".

Adding a fact is easy. We just take the name associated with the query, and index it into the map, and insert into the deque that is associated with the name.

To Inference a simple fact from the Knowledge Base, for example INFERENCE Father($X, John), the class method first obtains the Deque<Query> associated with Father, and iterates through the whole Deque looking for Query Objects whose parameter list is the same size as the inference. After this first round of filtering, we then iterate through all query object, and for every query object, we put the parameters in a temporary map<string, string> varMap.

After processing this query's parameters, we then match with the map the inference query to see if there are differences in the varMap map. For example, to match (Bob, John) to ($X, John), we take varMap[Bob] = $X, and varMap[John] = John. If we see that the keys match or are different in the case of holding a variable, we let this query object into the return deque to the caller. However, if we find that the same variable is assigned to two different keys we know to not match this query, and we don't let it into the output deque.

To remove a fact from the Knowledge Base, the class calls QueryFact to obtain a list of queries that the user wants to remove. Then it matches all Query objects to the knowledgeContainer for removal. This way the user can delete multiple facts using variables. This function also removes a whole name if the user so chooses to remove all Fathers, or Apples, by keying into the map and clearing the deque.

IV. Rule Base Testing and Implementation

The Rule Base has 3 main functions:

- **Int AddRule(Query query)** - Checks if a rule is already defined in the rulebase and if it isn't then insert the query object into the deque.
- **Int RemoveRule(Query query)** - Goes to the key value and clears the deque associated with it.
- **Int QueryFact(Query query, deque<Query>& output, KnowledgeBase kb)** - Checks the rule operator for the rule first.
    - OR operator -- Will spawn 2 threads which look up knowledge base in parallel :
        - Check if there is a rule with the name of the left half of the rule.
        - If there is a rule then recursively call QueryRule on that rule to break it down
        - If there is no rule then create a fact query by copying the input parameters into the search query and call QueryFact in the knowledge base.
        - Do the same for the right side of the rule on another thread
    - AND operator :
        - Thread 1 : AndRuleL:
            - Swap the first parameter of the left side of the rule with the first parameter of the inference
            - Check if there is a rule with the name of the left half of the rule
            - If there is a rule then recursively call QueryRule with the swapped paremeters
            - If there is no rule then call QueryFact and get all the results with the custom inference query
            - In each recursive call, push a matching query to a global buffer shared with Thread 2.
        - Thread N (Multiple can be spawned): AndRuleR
            - Grab a query from the global buffer.
            - Loop through the output of the left half of the rule and replace the the matching parameters between the two halves of the rule and call QueryRule or QueryFact depending on if a rule exists, with the query grabbed from global buffer.

        - Concatenate the results from both half so that we get an output that matches the inference called.
        - Push results to output buffer
    - Calling Thread: Waits for Thread 1 and N to finish. Compile results to a deque and send to interface for printout.

Implementing the idea for the OR operator was simple and easy to come up. The OR operator searches for the results of both halves of its rule definition. Because of that you can just copy the parameters being inferenced and insert it as parameters for both of the rule/facts to search for. The two parts of a rule can be made of a rule or a fact and rules break down into more facts/rules. The fact that a rule can be made of multiple rules going down for an unknown amount makes it hard to guess how to best handle multiple rule queries. We solved this by using a recursive call on QueryRule because our query objects can just be used to call on more rules and this will go on until it hits the case of the rule being made up of facts to search.

Implementing the AND operator was significantly harder and more complex because we originally wanted to keep the idea of using multiple parameters, but stuck with only two parameters because the cases got too complex to deal with. The base idea is that an inference would consist of two variables $X and $Y which would be the result. The first half of the rule only needs to be called as if it was the operator. No special actions was needed. This would work, but we also needed to cover the case of having an inference that deals with searching for a specific person, so the first parameter of the inference is swapped in the rule/fact query to keep it consistent. The second parameter of this fact/rule query is kept as a variable $Z because it is the matching variable between both halves of the rule and does not relate to the overall inference query. The output of this query would store all of the results into a temporary deque to be searched through soon.

The second half of the AND operator became tricky because of the logic behind swapping the parameter from the output to the input of the next part of the rule. Our implementation works by looping through all of the output from the first query made and swapping the second parameter from the facts retrieved and swapping them into the first half of our new query. We then take this new query and search through the rule/knowledge base again. The results retrieved by the second half of the query are the results of the entire inference. The results need to be matched with the original query that called it so we use a map to find the matching results from the first rule query called and store the index it is stored in the temporary output. Then we loop through the output of the second rule query and call the map to find the index of the matching query that has the first

parameter we need for our output. We create a new query object with the name of the inference and push into a deque the first parameter of the first rule query output and the second parameter of the second rule query output and this will be the output of the entire inference. The query is pushed into another deque to be returned as the output and printed out.

V. Interface Testing and Implementation

From the parsed string the interface class takes care of what functionality to execute for the Rulebase/Knowledgebase system. The parsed string is stored in a struct named query and in the struct there is a string called command that holds the the function the user wants to execute.

ExecuteCommand(string command) compares the query command with a the different possible commands the program implements, if it matches it calls its respective function in the program from the QueryOperations class.

Load(Query query) is a function that gets called if the string command was LOAD when running executeCommand runs. It opens up the passed file, reads and for loops through each line to call execute command on that line. We included Load in the this class instead of QueryOperations because it uses the executeCommand.

Print(deque<Query> queries) prints the results of the inference command. The results of the inference is stored in a deque of queries. The print function loops through the deque and prints out each index.

QueryOperations class contains the functionality of the commands that are parsed and passed into the Interface::executeCommand(string command). It contains private objects to the KnowledgeBase, RuleBase and Dumper.

Add(Query query) is a function that adds either a fact or rule to it's respective place. The function has if statements that checks the "ident" string in the query struct which says whether is a fact or rule. If it's a fact it calls add Fact in the knowledge base. If it's a rule it calls AddRule in the rule base.

Remove(Query query) removes the Fact or Rule from both the knowledge base and rule base. This is done because in case a rule and fact exist it's supposed to remove from both.

Inference(Query query, deque<Query>& output) inferences the query to print the facts for that inference. I checks if the fact exists, if it does, it querys the fact from the knowledge base. It does the same thing for the rulebase. It checks both for example in case the user defines a fact as grandmother but also define grandmother as a rule made up simpler facts. So if a grandmother isn't explicitly

defined between strings. Grandmother might still exist if it goes through the rule based and deduces it through the facts in the knowledgebase.

Dump(Query query) just calls the dump function in the dumper class to edit the buffer string that the client will write to a file.

VI. Dumper and Loader, Testing and Implementation

The Dumper's parsing is very specific and in depth, because it needs to convert query instances into perfectly formatted commands that would result in said query. By contrast, the Load function does no parsing, and simply feeds each line of a given file to the executeCommand function. The first iteration of the Dumper's parsing algorithm was *al*most perfect, but had a small problem with how it printed rules. This was caught by loading from a properly formatted file, dumping to a new file, and then comparing the two. Once the problem with empty inputs (described below) was fixed, the load function had no further issues.

VII. Socket Interface

We implemented the socket class from Prof. Karim's slides and Ecommons file distribution. The connection class was built by looking at the examples during lecture. We used POSIX Sockets which will work on linux. We encountered a few bugs, which we had to flush stdout to get output on the command line.

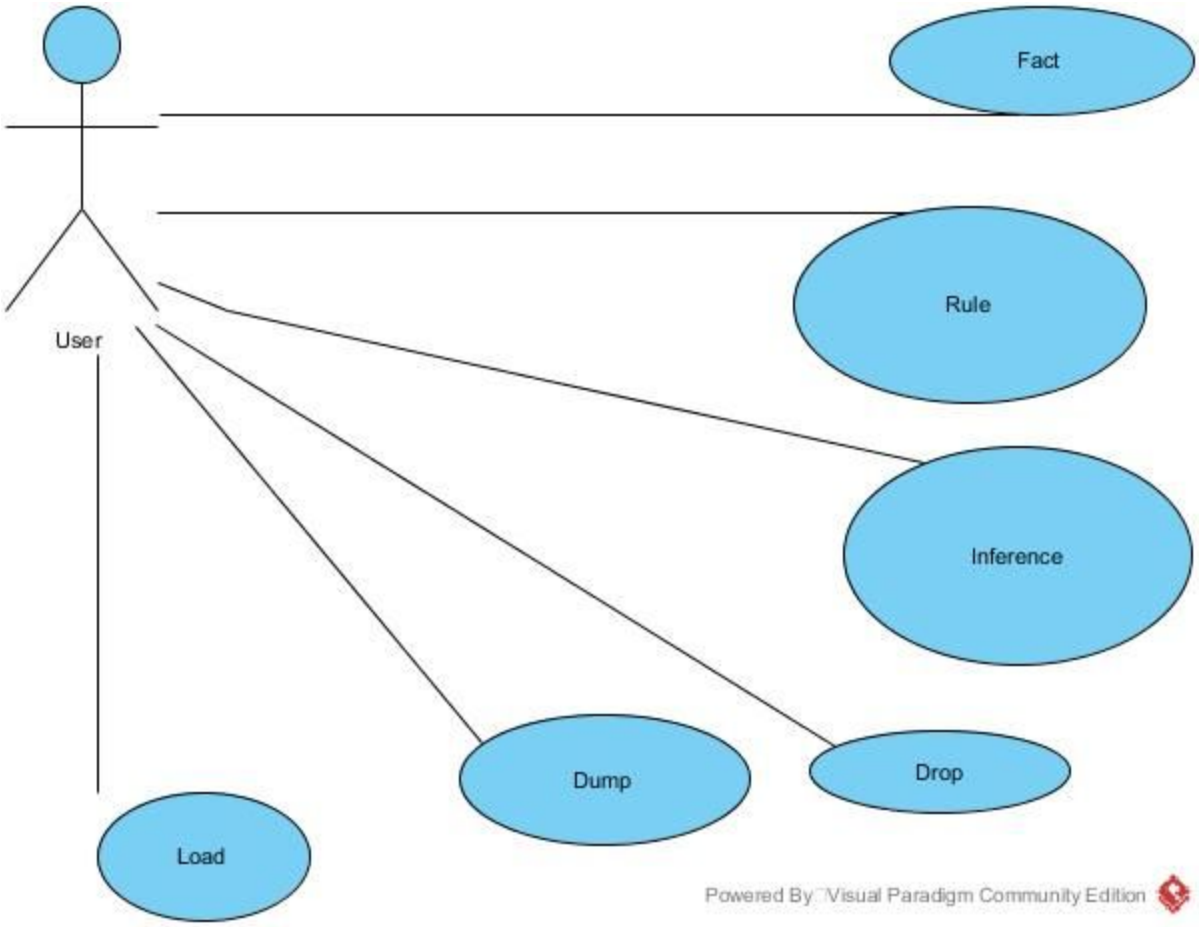
We only tested by having the server and client on same localhost. Refer to assumptions section.

VIII. Edge Cases Covered, and Anomalies Encountered

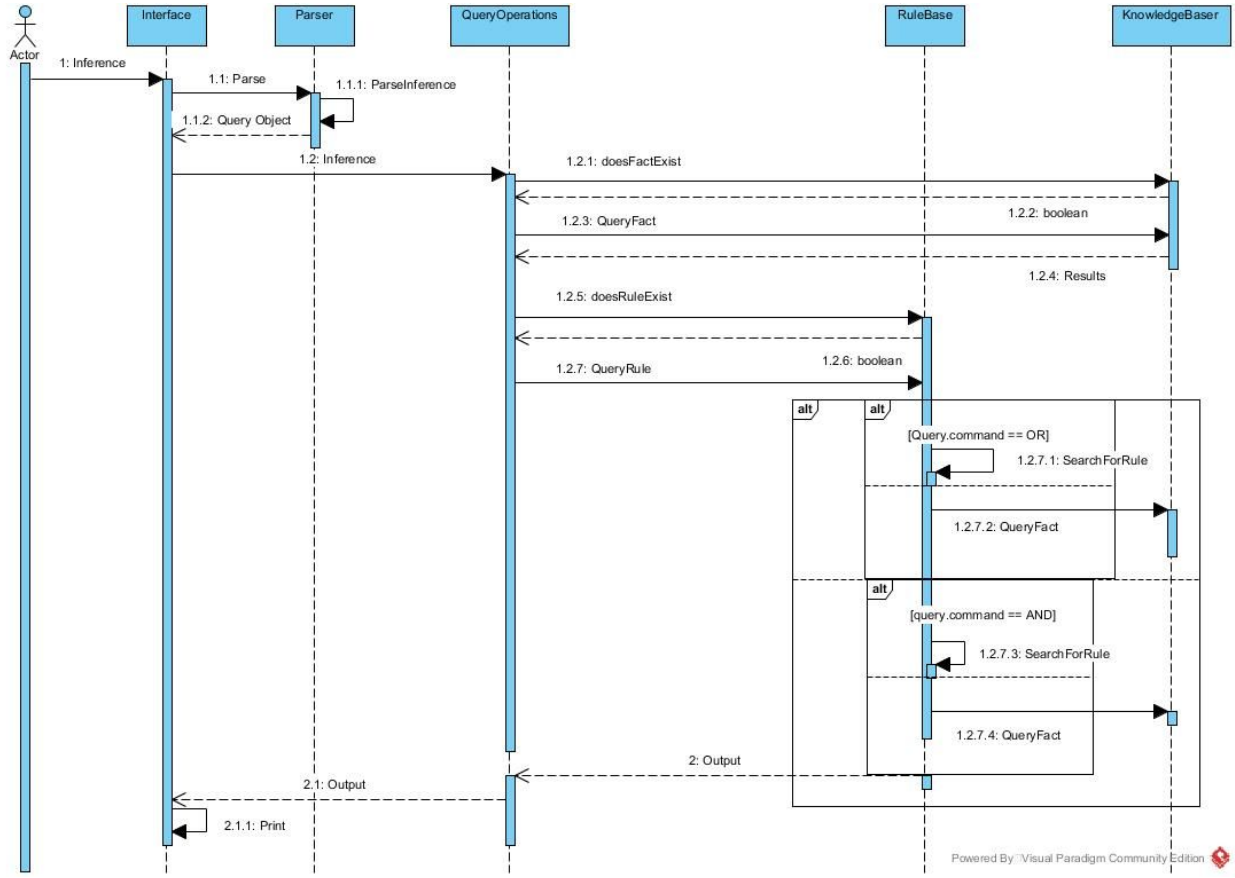
For a while, we encountered segmentation faults whenever we gave the parser input that consisted of only spaces, commas, or parameters. This happened because the parser split its input string into a vector, using those three characters as delimiters. The resulting vector would have 0 elements in such cases, so in the following line, a reference to vector[0] caused the fault. We fixed this by checking that the length of the vector was at least 1, throwing an exception if it was not, and catching the exception at a higher level and handling it by cancelling the operation and printing an error.

UML Sequence Diagrams

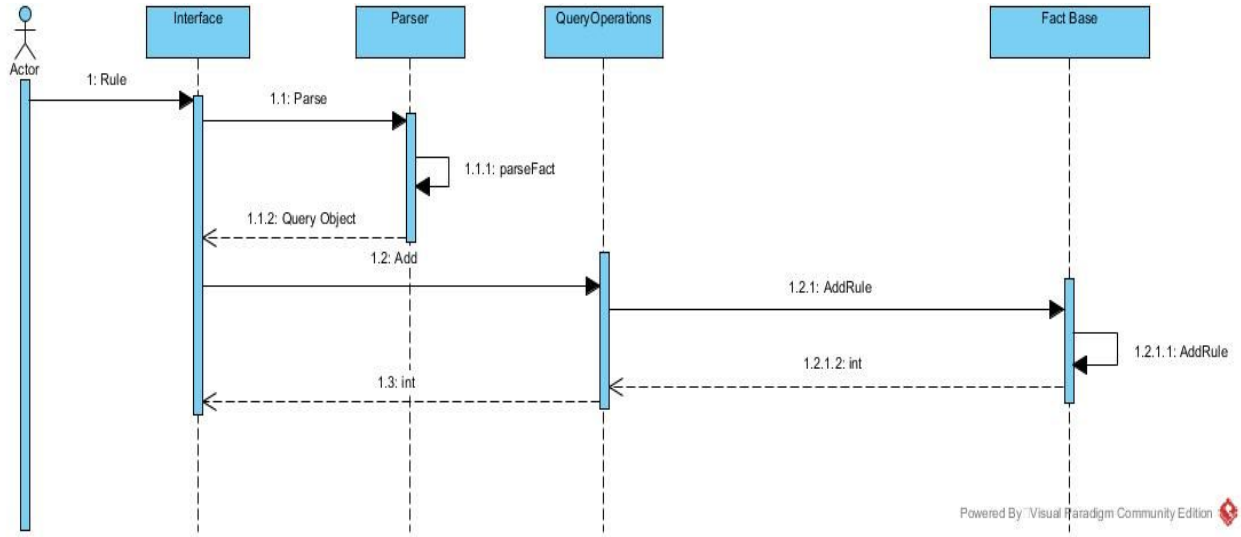
These diagrams show how a user will interact with the program. Actor will represent the front end of the SRI Server. SRI Client sends plaintext to SRI Server. SRI Server Frontend transfers the plaintext from client to SRI Engine (Back End) -- thus the front end is the actor.


Fact
Rule
User
Inference
Dump
Drop
Load
Powered By Visual Paradigm Community Edition

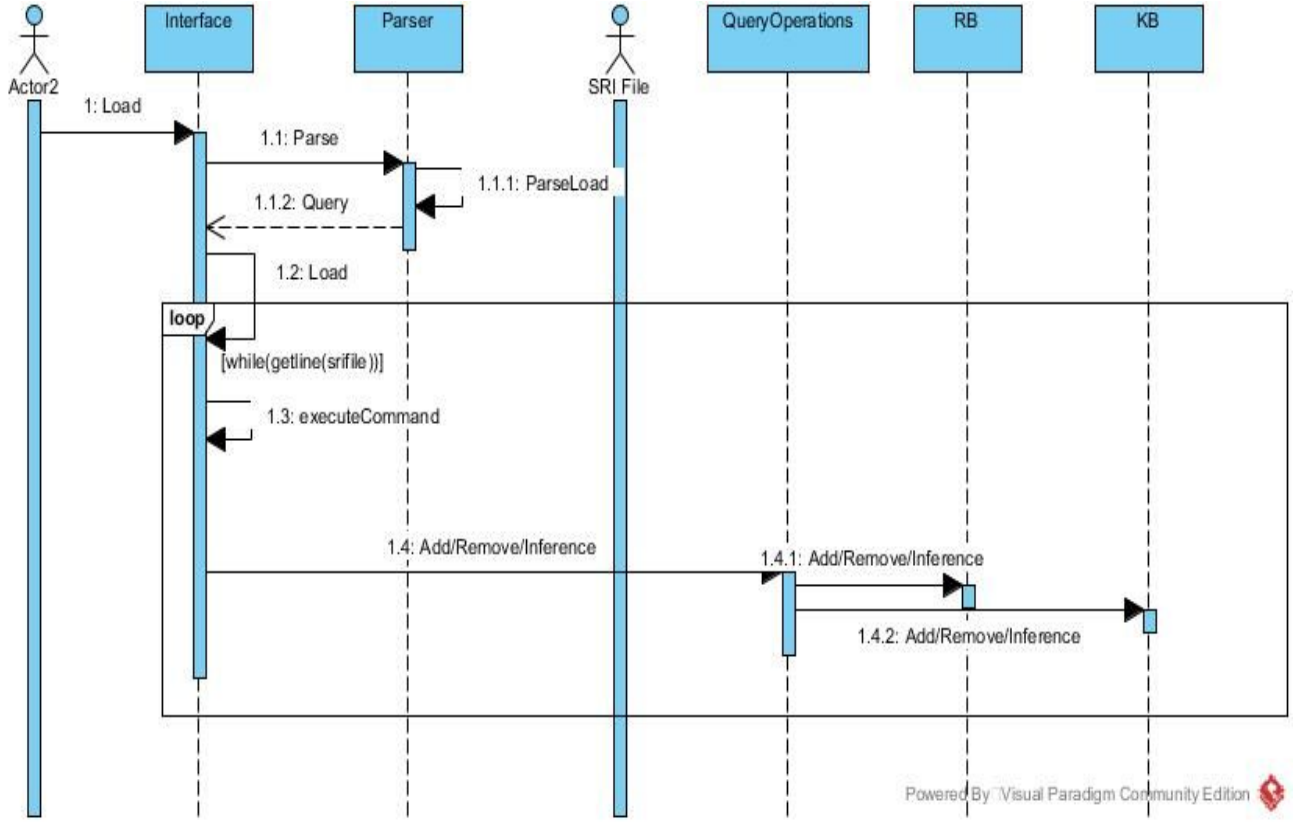
Inference Sequence



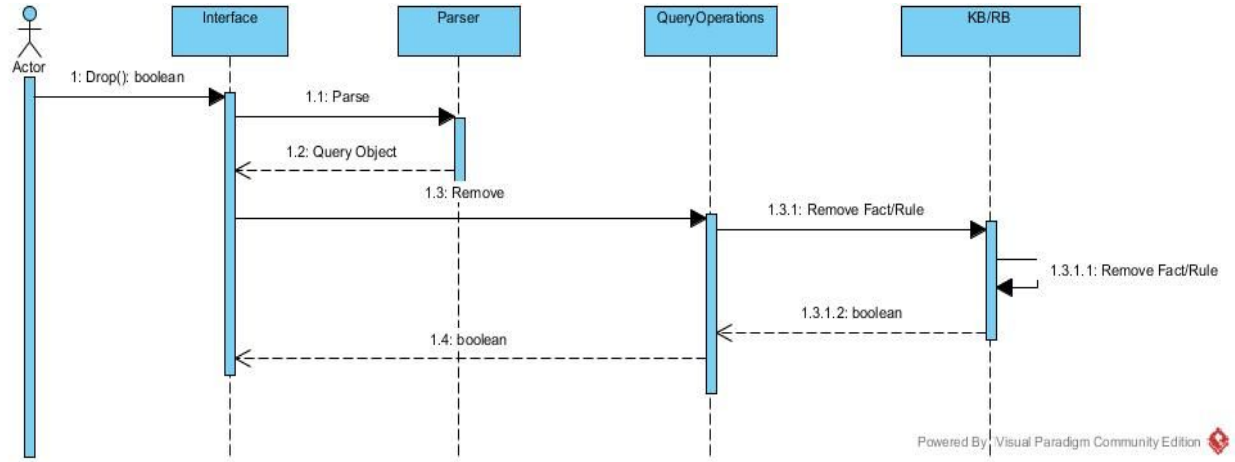
RULE Sequence



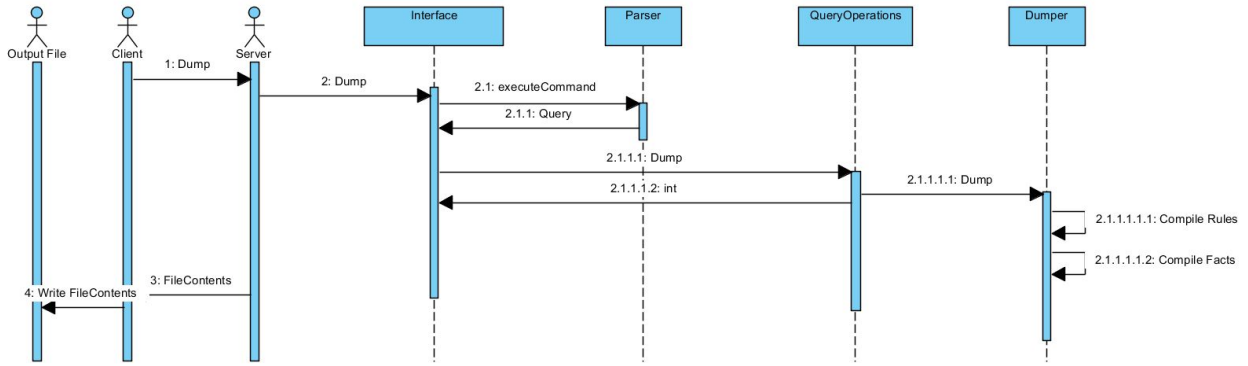
LOAD Sequence



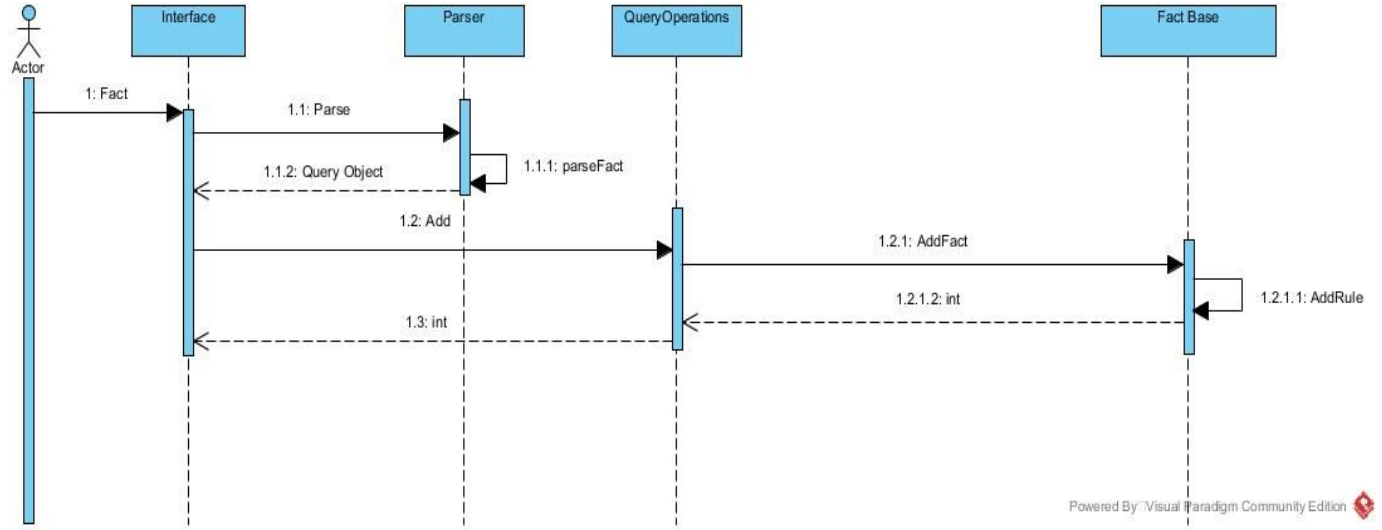
DROP Sequence



DUMP Sequence



ADD FACT Sequence


Actor
Interface
Parser
QueryOperations
Fact Base
1: Fact
1.1: Parse
1.1.1: parseFact
1.1.2: Query Object
1.2: Add
1.2.1: AddFact
1.2.1.1: AddRule
1.2.1.2: int
1.3: int
Powered By Visual Paradigm Community Edition