UNIVERSITY OF DUBLIN, TRINITY COLLEGE

Memory Efficient Stream Reasoning on Resource-Limited Devices

by

Dylan Armstrong

Supervisor: Declan O'Sullivan
Assistant Supervisor: Wei Tai

A dissertation submitted in partial fulfillment for the
degree of Master in Computer Science

Submitted to the University of Dublin, Trinity College, May 2014

Declaration of Authorship

I, Dylan Armstrong, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signed: _____

Date: _____

Summary

The first goal of this research was to determine which of two reasoning algorithms could be adapted for use in a streaming reasoning system with memory constraints. The two algorithms investigated were TREAT and LEAPS. Once achieved, the primary goal was then to determine whether an implementation of the chosen algorithm provided a more efficient option in terms of memory consumption in a resource constrained device in comparison to an existing implementation of the RETE algorithm.

This paper first illustrates the motivation for this research by providing context for why and when memory constrained devices are used in reasoning. Standard approaches usually attempt to perform reasoning on a centralised machine, sending data from sensor devices through the web or by other wired or wireless technologies. However, in some conditions this may not be feasible, or may be restrictive. In these cases reasoning can be performed on the sensor device nodes themselves. For this reason, the reasoning process must be memory efficient.

Previous work has seen the adaption of a static RETE based reasoner (COROR) into a stream based reasoner (SCOROR). A streaming environment is commonly the sort of environment that memory constrained sensor devices are used in. This research focused on finding an alternative reasoning algorithm for SCOROR that could offer improved memory consumption.

A background on reasoning, both static and stream, is given. A background on SCOROR is also provided so that the stream reasoning process can be understood in more detail. The RETE, TREAT and LEAPS algorithms are explained in detail. A comparison is offered, with LEAPS proving the most promising for improved memory consumption.

An implementation of the LEAPS algorithm was completed and integrated into SCOROR so that comparisons to the RETE reasoner could be performed. An in-depth design and implementation of the LEAPS reasoner is given, along with the steps needed to modify the algorithm in order to support stream reasoning.

Finally, an evaluation was performed in order to compare the existing RETE reasoner with the newly implemented LEAPS reasoner. The evaluation found that the reasoning times for both reasoners were quite similar. The RETE reasoner outperformed the LEAPS reasoner when there was a combination of a large ontology size with high expressivity. The suggested reasons for this are the use of wild-card predicates in rules and the

resetting of saved iteration states due to the removal of temporal triples. Memory consumption for the LEAPS reasoner was shown to be significantly reduced in comparison to the RETE reasoner.

This research shows that the LEAPS algorithm can be implemented and modified to perform in a streaming environment. Evaluation suggests that it is indeed the case that a more efficient option in terms of memory consumption over the RETE implementation exists in the form of this implementation.

Acknowledgements

I would like to thank both Declan O'Sullivan and Wei Tai for their guidance and assistance during this research. I would also like to thank Colin Hardy for taking the time to explain the operation of SCOROR to me.

Finally I would like to thank my family and friends for their continued support through my entire college experience.

Dylan Armstrong.

Contents

List of Figures

List of Tables

Abbreviations

BRMS	**B**usiness **R**ule **M**anagement **S**ystem
C-SPARQL	**C**ontinuous-**SPARQL**
CE	**C**ondition **E**lement
COROR	**CO**mposable **R**ule-entailment **O**wl **R**easoner
DO	**D**ominant **O**bject
LIFO	**L**ast **I**n **F**irst **O**ut
LEAPS	**L**azy **E**valuation **A**lgorithm (for) **P**roduction **S**ystems
OWL	**W**eb **O**ntology **L**anguage
RDF	**R**esource **D**escription **F**ramework
RDFS	**RDF S**chema
SCOROR	**S**tream **C**OROR
SPARQL	**S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage

Chapter 1

Introduction

1.1 Motivation

In sensor-rich systems it is common for sensor data to be gathered from a multitude of separate sources, for example in environmental observations [1] and hazard assessment [2]. This data is often difficult to categorise and interpret. Semantic Web technologies have been a major source of interest in sensor-rich systems as a result of this [3] [4]. These technologies offer methods to aggregate and interpret this data. Standard approaches usually attempt to perform these operations on a centralised machine, sending data through the web or by other wired or wireless technologies [1] [2].

However, in some conditions this may not be feasible. For example, consider monitoring that may take place under water or in extreme weather conditions. Environmental conditions are subject to dramatic change and can cause interference between nodes in the network. Consider also the bottleneck that may arise in a system when hundreds of sensors are attempting to push data to, and receive data from, the centralised node. For these, and other reasons, it is not unheard of to manage data reasoning on the data sensor devices themselves [5] [6].

When devices are resource constrained any algorithms that are to be used must be extremely memory efficient. Previous semantic reasoners have been considered too exhausting of resources for use on resource constrained devices [5] [7] [8].

Attempts have been made to implement the RETE reasoning approach in a resource constrained environment [5], but it may be possible to achieve further memory reductions with the implementation of other algorithms, namely TREAT [9] or LEAPS [10].

TREAT is an optimised version of RETE in terms of memory consumption, but it is known to sacrifice speed in some circumstances. It attempts to reduce memory usage by storing a reduced amount of state information in between reasoning cycles.

LEAPS is a lazy evaluation reasoning approach. While RETE and TREAT both attempt to find all solutions for a proposed question, LEAPS simply attempts to find the first possible solution. It is known to offer linear space complexity, while it's predecessors, RETE and TREAT, offer exponential space complexity.

Previously mentioned research has mainly focused on static reasoning. To fully understand and interpret data in sensor rich systems, reasoners must have the ability to quickly and efficiently reason over high frequency data while considering background knowledge [11]. The need for stream reasoning has been acknowledged in smart cities [12] and robotics [13], in order to support situation awareness, execution monitoring and decision making.

When the data set that needs to be processed is being updated with high frequency, i.e. data coming from a stream, and this data needs to be processed quickly, a stream based reasoning system is necessary. Recent work has seen the adaption of a stream based reasoning system for use on resource constrained devices [14]. This system is an expansion of the RETE reasoning approach implemented by Tai et al [5].

If either TREAT [9] or LEAPS [10] are to be adapted as a memory reducing replacement for RETE, the chosen algorithm must be modified in order to support stream reasoning.

The first goal of this research is to determine which of the two above algorithms are more suitable for use in a streaming system with memory constraints. Once achieved, the primary goal is then to determine whether an implementation of the chosen algorithm provides a more efficient option in terms of memory consumption in a resource constrained device in comparison to an existing implementation of the RETE algorithm.

1.2 Research Question

> "In comparison to an existing implementation of the RETE algorithm, can a more efficient approach, in terms of memory consumption, be developed for stream reasoning on resource constrained devices?"

1.3 Dissertation Overview

A brief description of the following chapters is given in this section.

1.3.1 Background

The intention of this chapter is to give the reader an understanding of the concepts associated with stream reasoning and the background, technologies and assumptions on which this research is based on.

1.3.2 State Of The Art

This chapter discusses the research performed in the area of reasoning algorithms. A description, analysis and comparison of relevant algorithms is performed.

1.3.3 Design

An in-depth description of the chosen algorithm is given in this chapter.

1.3.4 Implementation

The implementation of the chosen algorithm is discussed in this chapter.

1.3.5 Evaluation

In this chapter an evaluation of the newly implemented algorithm is performed. Results relating to the reasoning performance and memory consumption of the new algorithm implementation are compared with the existing algorithm implementation.

1.3.6 Conclusion

Finally, a conclusion on the research and results found is offered.

Chapter 2

Background

2.1 Introduction

This chapter aims to give the reader a basic understanding of the field of research by providing information on concepts associated with stream reasoning and the background and assumptions on which this research is based on. Reasoning, stream reasoning and the components of SCOROR, an existing stream reasoner, are discussed.

2.2 Previous Work

The research described in this work is based on initial work performed by Tai et al [5] and a RETE based reasoner called COROR that was developed in conjunction with their research. It is also based on research performed by Hardy [14], who converted this reasoner into a stream based reasoner called SCOROR.

2.3 Reasoning

In this work reasoning is informally defined as going beyond the information that is given. Reasoning systems are those which have the ability to take facts and rules as inputs and come to some conclusion without the need for human input. Semantic reasoners are commonly used in the Semantic Web. Many different categories of reasoners exist

including rule-entailment, tableau and resolution-based. This research deals with rule-based reasoning, where a rule is composed of at least one "if-then" condition element and a resulting action. This action is performed if the condition elements of the rule are met. Rules are contained in a rule set. Examples of rule-based algorithms include RETE [15], LEAPS [10] and TREAT [9].

A reasoner generally draws facts from an ontology into a working memory. Facts in the working memory are tested against the rule set. Performing the action of a rule may result in further facts being inserted into the working memory, or indeed facts being removed from the working memory.

A good analogy to convey the operations of a reasoning system is given by Miranker [9]. Miranker makes use of relational database terminology. The following is an adaption of that analogy.

> "If the working elements of a reasoning system are considered to be tuples of some universal relationship in a relational database, then it becomes apparent that the condition elements of a rule in a reasoning system is analogous to a query in a relational database language. The constants in a single-condition element may be viewed as a relational selection over a database of the working memory. We say a working memory element partially matches a condition element if it satisfies the select operators or the intra-condition element pattern constraints. Consistent bindings of pattern variables between distinct condition elements may be regarded as a database join operation on the relations formed by the selections. The conflict set is the union of the query results of each of the rules in the system."

The three previously mentioned algorithms can also be classed as forward-chaining reasoning algorithms. Forward-chaining indicates an incremental procedure, where rules and facts are used to deduce new facts. In turn, these new facts are used to deduce further facts. This process continues until no further facts can be deduced by the current rule set and the current set of facts in the working memory.

The alternative of forward-chaining is backward-chaining. Backward-chaining algorithms, implied by the name, usually work backwards in order to satisfy a goal. In

this case, the goal is to find facts that satisfy a particular outcome of a rule. Therefore a query is needed to perform backward-chaining.

There are advantages and disadvantages for both approaches. Forward-chaining offers quicker delivery of results at query time due to the fact that all deductions have already been produced. The disadvantage of this approach is the amount of memory used to hold these deductions. In contrast to this, backward-chaining produces deductions at querying time resulting in minimal memory usage. The disadvantage here is that reasoning must occur at query time, resulting in a much slower response.

Reasoners are designed to have a specified ontology language, usually defined generally by a frame language. A specific mark-up language (OWL [16], RDF [17], RDFS [18]) is then used to implement general concepts defined by the frame language. OWL, RDF and RDFS are based on description logic. These languages have been used frequently by researchers in recent times, as they are used regularly in the Semantic Web. Indeed, these are the languages used by the reasoner implemented in SCOROR.

The W3C created the specification of Resource Description Framework (RDF) which was then built upon by RDF Schema (RDFS) and Web Ontology Language (OWL) to create a more complete model for structuring data in a way that could be reasoned over.

2.3.1 RDF Structure

RDF structures data as triples. Triples consist of a subject, object and predicate. The predicate is a relationship that relates the subject to the object. For instance if a class A is a sub class of a class B it is possible to create a statement to represent this information by using the subject A, the predicate "rdfs:subClassOf" and the object B. OWL adds the ability to give the data implicit knowledge by offering predicates that model relationships such as transitive and inverse properties.

In this research, a fact is then simply a triple. A rule is composed of at least one condition element. A condition element is modelled as a triple, where the subject, predicate and object have specific values. In a condition element it is also possible that the subject, predicate and object are variables, and may be satisfied with any value.

An example of a RDF rule is given below. The left hand side represents two condition elements for this rule. The right hand side represents the action. The question marks represent variables.

$$(?a \;\; rdf{:}type \;\; ?b),\ (?b \;\; rdfs{:}subClassOf \;\; ?c) -> (?a \;\; rdf{:}type \;\; ?c)$$

In this case, if a triple exists in the working memory that has the predicate "rdf:type", then that condition element of this rule is considered to be satisfied. If a second triple exists in the working memory that has the predicate "rdfs:subClassOf", and the object of the first triple matches the subject of the second triple, then this condition element of the rule is considered to be satisfied. As both condition elements are satisfied, the rule is satisfied. The action of the rule is then performed. In this case, this results in a new fact being added to the working memory. This new fact is composed of the first fact's subject as the subject, the second fact's object as the object and the new predicate "rdf:type".

2.3.2 Introducing State

To return to the relational database analogy put forward by Miranker [9], a database system usually computes a single query at a time over a relatively large database. It is possible to think of a reasoning system as computing many queries, as many as there are rules, over a slowly changing, modest size database.

In order to reduce the recalculating of comparisons that may occur on different cycles, reasoning systems usually retain some state across cycles. Three types of state information which are commonly found in reasoning systems have been identified by McDermott, Newell and Moore [19]. These are listed below.

1. Condition Membership: Associated with each condition element in the reasoning system is a running count indicating the number of working memory elements partially matching the condition element. A match algorithm that uses condition membership may ignore those rules that are inactive. A rule is active when all of its positive condition elements are partially satisfied.

2. Memory Support: An indexing scheme indicates precisely which subset of working memory partially matches each condition element. By analogy, memory support systems explicitly maintain a representation of the relations resulting from the select operations.

3. Condition Relationship: Provides knowledge about the interaction of condition elements within a rule. By analogy this corresponds to explicitly maintaining the intermediate results of a multi-way join.

2.4 Stream Reasoning

In this research stream reasoning refers to the means of reasoning over a continuous flow of data. While reasoning in general and the RETE reasoner described by Tai et al [5] usually deals with a static knowledge base of facts, stream reasoners deal with a rapidly changing knowledge base. Hardy [14] expanded on the work carried out by Tai et el to convert their reasoner into a stream reasoner.

It is not feasible to store all data that arrives on a stream as this could overload a fast complex system quite easily. Alternatively, it does not make sense to store a single item of information without context of the other data in the stream. For this reason, a window based approach is usually used to limit the data that is inserted into the reasoner at a given time.

As has been mentioned before, incremental reasoning is based on re-using the deduced data to continuously generate more data. The process is beneficial as it reduces reasoning time by using previous results. With stream reasoning there is a continuous flow of data being reasoned over and consequently this would result in an ever expanding amount of data being held in the reasoners working memory and inner data structures. Removing older data frees computing resources, i.e. memory and processing power, to react to newer information in a timely manner. Further, in streaming real-time environments it is often assumed that older data becomes irrelevant as time goes on [11].

Hardy implemented a window based approach using C-SPARQL that sets an interval on the stream. At a given time, whatever items are within the interval are considered valid. The items are then added to the working memory of facts and the normal reasoning

approach can be applied. These facts are referred to as "temporal facts" and are marked with a time-stamp. Consequently, any deduced facts are also marked with a time-stamp. After a certain period these temporal facts are removed from the working memory and reasoner data structures to ensure that stream reasoning may continue.

2.5 C-SPARQL

SPARQL [20] is the standard querying language that allows for the retrieval and modification of data that is stored in the RDF format. It provides a means to extract meaningful information from large collections of data. Is is akin to SQL for relational databases.

A simple SPARQL query consists of a set of triples where the subject, object or predicate may be variables. The result of the query would be any existing RDF triples in the ontology that match the input triples and that satisfy any variables.

C-SPARQL [21] (Continuous-SPARQL) is an extension of SPARQL that provides the ability to perform continuous queries over complex streams of RDF data. C-SPARQL queries allow the user to specify a window over the stream of data. A window can be configured so that it returns all triples within a specific time frame (a logical window) or so that it simply returns a set number of triples (a physical window). In the case of a physical window, the set of triples returned is simply the last N triples which have arrived on the stream, where N is the size of the window. Once the set of triples has been determined, normal querying is performed using other parameters provided in the original query.

An example C-SPARQL query taken from [22] is shown in Figure 2.1.

```
REGISTER QUERY GlobalCountOfInteractions
        COMPUTED EVERY 5m AS
SELECT ?user COUNT(?document) as ?numberOfMovies
FROM STREAM <http://streamingsocialdata.org/
     MoviesJohnsFriendsLike.trdf> [RANGE 30m STEP 5m]
WHERE { ?user sd:likes ?document }
GROUP BY { ?user }
```

FIGURE 2.1: Example C-SPARQL Query

This is a simple query used in a mock social networking scenario, with a user "John". The query counts, among the friends of John, the number of movies that each friend

has liked in the last 30 minutes. The stream provided is simply a list of triples that represents movies that John's friends have "liked". This query is executed every five minutes, and makes use of a logical window to extract any triples that have been added to the stream in the last thirty minutes.

2.6 RETE

The implementation of the RETE algorithm in a stream reasoner forms the basis of inspiration for this research [14]. The algorithm is considered efficient in terms of speed, but further memory optimisations are desirable.

The RETE algorithm makes use of both memory support (Alpha networks) and condition relationship (Beta networks). In its fundamental form, the RETE algorithm can be explained as follows. For each rule that the data is to be tested against, a tree like matching network (referred to as a RETE network) is created. This network is itself separated into two networks, the Alpha and Beta network. The Alpha network is composed of Alpha nodes which represent the condition elements of the rule. For example, consider the rule below.

$$(?a \;\; rdf{:}type \;\; ?b),\ (?b \;\; rdfs{:}subClassOf \;\; ?c) -> (?a \;\; rdf{:}type \;\; ?c)$$

This rule would result in the creation of the Alpha nodes shown in Figure 2.2.

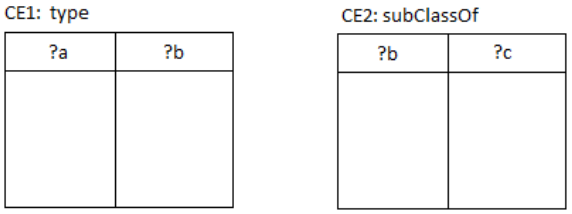

FIGURE 2.2: Alpha Nodes

Now consider we have two facts:

$$(Ford \;\; rdf{:}type \;\; Car)$$

$$(Car \quad rdfs{:}subClassOf \quad Vehicle)$$

When a fact is passed into the RETE network it is referred to as a token. Each Alpha node stores the tokens that satisfy that particular condition element, thus forming the memory support part of the algorithm.

This would be represented in the Alpha network as seen in Figure 2.3.

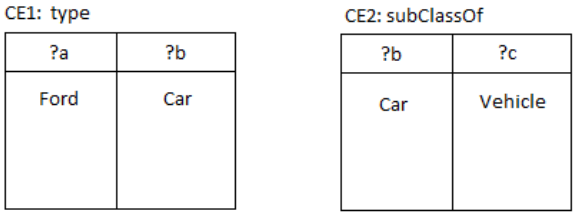

FIGURE 2.3: Alpha Nodes with values

Following the Alpha network are two-input test nodes that test for consistent variable bindings between condition elements. These are referred to as Beta nodes and combined they form the Beta network.

The Beta network joins input tokens into combinations stored in Beta nodes. Beta nodes can have inputs composed of tokens, or of a token and an existing Beta node. Inputs with consistent variable bindings are then stored in the Beta node.

The above rule would result in the creation of the Beta node shown in Figure 2.4.

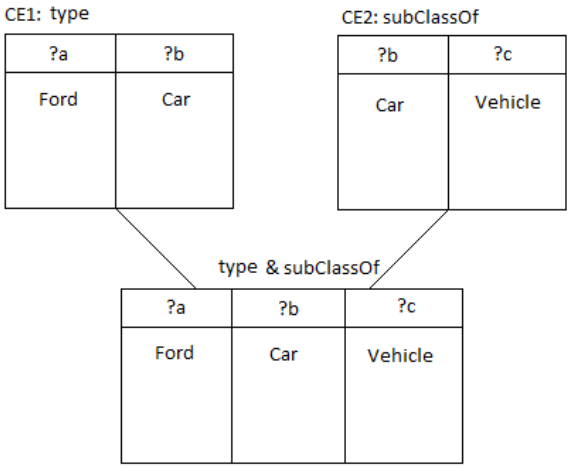

FIGURE 2.4: Alpha Nodes with Beta Node

When a combination propagates from the last Beta node, it must satisfy all the conditions of a rule. This is called a "rule instantiation". The instantiation is then added to the conflict set. The conflict set is a set of the rules which are scheduled to perform their action. A rule performing its action may result in the addition or removal of facts in the working memory. In our example, the new fact below would be added to the working memory.

$$(Ford \;\; rdf{:}type \;\; Vehicle)$$

RETE attempts to take advantage of two empirical observations.

1. Temporal Redundancy: The firing of a rule usually changes only a few facts, and only a few rules are affected by each of those changes.

2. Structural Similarity: The same condition element often appears in the left-hand side of more than one rule. Structural similarity means that the algorithm can take advantage of node sharing, reducing memory consumption.

2.7 SCOROR

As the RETE algorithm and stream reasoning have both been introduced, it is now possible to describe SCOROR in more detail. SCOROR consists of two main components, namely the Stream Processor and the Reasoner component itself (COROR), see Figure 2.5 taken from [14]. There is also a Stream Generator. The Stream Generator is a simple component that generates triples that are appropriate for the current base ontology. The triples are placed onto a stream with a speed that can be specified by the user. It is important to note that the Stream Generator is purely used to generate streams for the testing of SCOROR and has no operation in the actual reasoning process. Therefore the memory usage of the Stream Generator is not included in the memory usage of SCOROR.

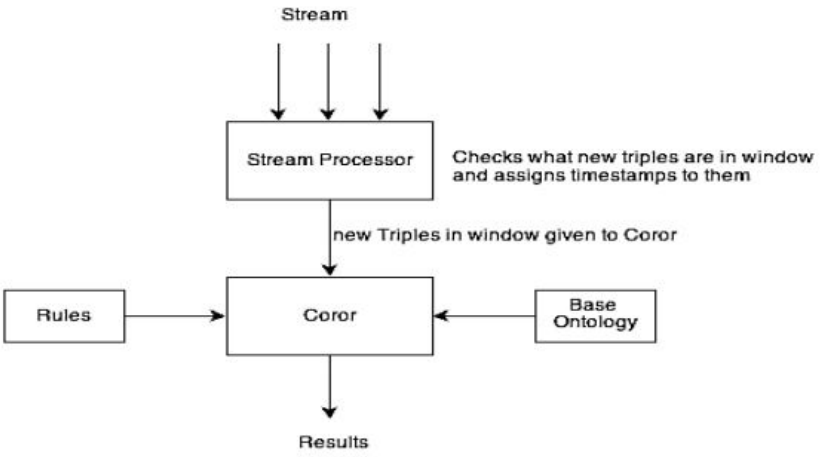

FIGURE 2.5: SCOROR Components

2.7.1 Stream Processor

The Stream Processor component is the component that makes use of C-SPARQL to query the incoming stream. As discussed earlier, C-SPARQL can be used to query the current triples on the stream. The results of this query compose the set of triples that will be inserted into the reasoner. These triples are assigned time-stamps before they are inserted into the reasoner, and are referred to as temporal triples once inserted. At the end of each reasoning cycle, the reasoner can then remove any temporal triples that have expired. This includes temporal triples that have been deduced due to the insertion of a now expired triple. The reasoner must also remove any associated tokens or other data held in data structures that is associated with expired triples. This time-stamp entailment approach to stream reasoning, which is used in SCOROR, was inspired by work by Barbieri et al [23].

SCOROR makes use of a C-SPARQL query that utilises a tumbling window query. Put simply, C-SPARQL returns all triples that arrive on the stream in 1 second intervals.

The Stream Processor then assigns time-stamps to these triples based on the current window size. It is important to note that this window size is separate from the notion of the C-SPARQL window. An example is given for a 5 second window below.

The reasoner is started and C-SPARQL returns all triples for the first second interval. These triples are passed to the reasoner at time 0s and so have the time-stamp

0+5=5s. The next interval is returned from C-SPARQL and these triples are passed to the reasoner at time 1s and so have the time-stamp 1+5=6s. At time 6, all triples with time-stamp 5s will be removed from the reasoner.

2.7.2 Reasoner

Reasoning as a whole has been discussed above. The reasoner in SCOROR makes use of a static ontology to first complete any initial reasoning. This static collection of triples is never removed from the reasoner and is used in conjunction with the incoming temporal triples to deduce further temporal triples. The current reasoner in SCOROR makes use of the RETE algorithm.

2.8 Summary

This chapter has explained what is meant by stream reasoning and has discussed the background work on which this research is based on. The individual components of SCOROR, and their role in stream reasoning, have been explained, including the RETE algorithm. The next chapter, State of the Art, discusses possible replacements for the RETE algorithm.

Chapter 3

State Of The Art

3.1 Introduction

This chapter performs an in depth discussion of two algorithms which could potentially replace the RETE algorithm in SCOROR. The algorithms are TREAT and LEAPS. Both algorithms are discussed and relevant advantages and disadvantages are offered in the hopes of determining which algorithm is more suitable for use in a streaming system with memory constraints.

3.2 Alternate Algorithms

The initial goal of this research was to determine whether an alternate algorithm could be used in place of RETE in SCOROR. The algorithm would need to be suitable for use in a streaming environment. The desired outcome was that an algorithm that offered a more efficient option in terms of memory consumption could be found. Two algorithms were considered for this task, TREAT [9] and LEAPS [10]. Both algorithms will be described and relevant advantages and disadvantages will be discussed.

3.2.1 TREAT

One of the algorithms which was considered for a resource constrained environment was the TREAT algorithm put forward by Miranker [9]. TREAT is an optimised version of

RETE in terms of memory, but it is known to sacrifice speed in some circumstances [24]. The motivation for TREAT arose when some observations were made about the RETE algorithm:

1. It is possible for some Beta nodes to redundantly store the same information.
2. Much of the information stored in the Beta network is also available in the conflict set.
3. A deletion of a fact from working memory is a costly procedure as it involves removing associated Beta nodes.

To illustrate the first point, consider the example discussed earlier in Section 2.6. The rule has been expanded with an additional condition element, as shown below.

$$(?a \;\; rdf{:}type \;\; ?b),\ (?b \;\; rdfs{:}subClassOf \;\; ?c),\ (?c \;\; rdfs{:}subClassOf \;\; rdfs{:}Resource)$$

$$-> \;\; (?a \;\; rdf{:}type \;\; ?c)$$

There is also a new fact to satisfy this condition.

$$(Vehicle \;\; rdfs{:}subClassOf \;\; rdfs{:}Resource)$$

RETE would produce the network shown in Figure 3.1.

Clearly there is a redundant store of data in the two Beta nodes that are generated by the rule.

The second point refers to the final Beta node in a network. This node holds the final data that makes up the instantiation. The data is then duplicated in the conflict set.

As a result of these findings TREAT does not generate Beta networks and attempts to exploit conflict set support. Conflict set support is an additional type of state information which was identified by Miranker [9].

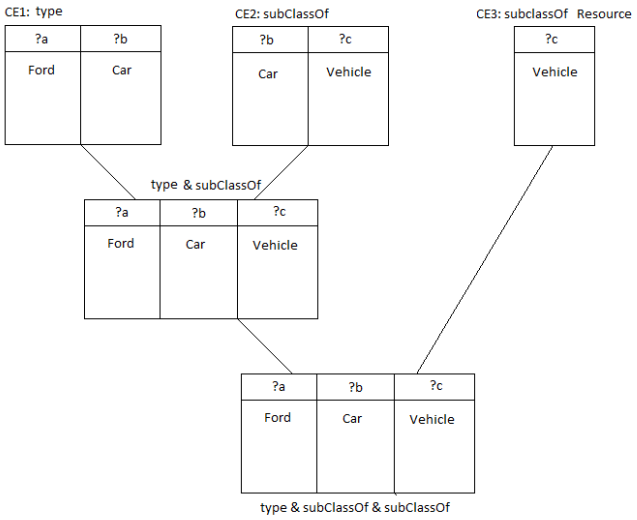

FIGURE 3.1: Extended RETE Network

- Conflict Set Support: The conflict set is explicitly retained across reasoning system cycles. By doing so, it is possible to limit the search for new instantiations to those instantiations that contain newly asserted working memory elements.

The algorithm still makes use of memory support (Alpha network), and also makes use of condition membership. To exploit conflict support, two observations must be made. It must be noted that these observations assume that negative rule conditions that test non-existence are not supported.

Firstly, if the action of a rule only results in the addition of a new fact to the working memory then the conflict set remains the same as before except for the addition of any new instantiations that contain tokens for the new working memory fact.

Secondly, if the action of a rule only results in the removal of a fact from the working memory then no new instantiations are added to the conflict set, but some instantiations may become invalid i.e. those that contain tokens for the deleted working memory fact.

This essentially means that additions to the working memory can be used as seeds in a constrained search to determine what are the new instantiations.

Deletions of facts from working memory are then straightforward, as the algorithm simply searches the conflict set for any instantiations that contain the data to be deleted and removes them from the conflict set. However, this means that an addition of a token

requires the full computation of all the joins that would otherwise have been present in the Beta network, resulting in a decrease in speed.

3.2.1.1 Negative Rule Condition Support

The addition of negative rule conditions that test non-existence pose a problem and complicate the algorithm.

Imagine a new token is added and instantiations that are based on the non-existence of this token are in the conflict set. The conflict set can't simply be searched for instantiations to remove, as tokens only represent the presence of data items. In these cases, the algorithm must treat negative conditions as positive, build the new instantiations and then remove any instantiations in the conflict set that match.

Imagine also the situation when a non-existence condition is satisfied by the removal of some token. Again the conflict set can't be searched, and TREAT must re-compute the intermediate Beta nodes to add any instantiations.

3.2.1.2 Condition Membership

Condition membership is suggested by Miranker [9] with the introduction of a unique Condition Element Number (CE-num) for every condition element in the reasoning system. A running count indicating the number of working memory elements partially matching the condition element can be retrieved from an array using the CE-num as an index.

If a rule has a condition element that has a CE-num of zero, the rule can be considered as inactive and thus does not to be included in the reasoning process.

3.2.1.3 Join Optimisation

When searching for consistent variable bindings between Alpha nodes the nodes can be considered in any order. This is due to the fact that the join operation is both commutative and associative. Miranker [9] describes an additional seed-ordering approach where recently changed Alpha nodes are considered first. This approach was favoured

above a static-ordering approach that considered Alpha nodes based on the lexical order of condition elements.

3.2.1.4 Comparison to RETE

The main advantage of RETE over TREAT is that the large amount of stored state data (Beta networks) drastically minimizes the chances of two working memory elements being compared. However, there are many disadvantages when using the RETE algorithm in comparison to TREAT:

1. When a fact is removed from working memory, the Beta network states must be unwound, often requiring the same operations that were required on its addition.
2. Beta memories may grow exponentially.
3. Join operations for the Beta network must be performed in a specific order, and this order must be determined statically at compile time.
4. Extensive computation may be performed for rules that may not be active with regard to the working memory facts.

With regards to the last disadvantage, Tai et al [5] developed a selective rule loader that determines what rules to include from the rule set by analysing the working memory facts. Unfortunately, this feature is not available for stream reasoning as it is not possible to know what rules will be required as the facts will be arriving by stream and are constantly being updated.

As discussed, the RETE algorithm maintains sufficient state to guarantee that no comparison of two working memory elements is recalculated at a later cycle. However, if large changes to working memory are made, a large overhead is incurred maintaining the state information. A system is considered temporally redundant if after each cycle proportionally small changes are made to working memory. The TREAT algorithm attempts to take advantage of this by constructing the Alpha networks and storing the conflict set between cycles, but ignoring the use of Beta networks. As the algorithm is to be used in a streaming environment, it is safe to assume that the system is not temporally redundant as the working memory will be continuously updated.

The key question surrounding TREAT is whether the number of comparisons performed by TREAT while searching for instantiations exceed the number of comparisons performed by RETE while processing deletions. The results of an empirical study performed by Miranker [9] indicate that TREAT outperforms RETE in this regard. It is also noted that this does not even consider the additional cost of maintaining the Beta networks.

Clearly the TREAT algorithm would be preferable in place of RETE as it offers better performance with regard to space complexity. With the additional seed-ordering approach, TREAT can also outperform RETE in terms of time complexity.

3.2.2 LEAPS

It has been claimed that the LEAPS approach is better in terms of time and space complexity than both RETE and TREAT [10] [24] [25]. LEAPS makes use of a lazy evaluation approach when reasoning and so does not fully enumerate the whole conflict set but instead processes a single rule instantiation per cycle. LEAPS actually began as a production system compiler for OPS5 rule sets based on research performed by Miranker et al [25]. The approach is based on both complex data structures and complex search algorithms. The analogy given by Miranker in Section 2.3 makes use of relational database terminology to describe the operations of a reasoning system. Unfortunately, the LEAPS algorithms have been known to be difficult to comprehend due to the inability of relational database concepts (i.e. relations and select-project-join operators) to capture critical lazy-evaluation features of the approach.

Batory [26] states how the approach is difficult to understand due to a lack of high level abstractions appropriate for expressing the details of the algorithms. However, the approach is described by Batory [10] in terms of the container-cursor programming abstractions of the P2 data structure compiler. It should be noted that even if the LEAPS algorithms had an elegant expression in a chosen language, the code would have to be very efficient to compete with the original LEAPS system.

An attempt is made below to describe the algorithm (examples taken from Batory [10]) in general programming terms and concepts.

3.2.2.1 Composite Containers

A container is simply a collection of elements of a single data type maintained in a sequence. Containers provide cursors which allow for the iteration over the elements within the container. The elements can be referenced and updated by the cursor.

A sample of a container would be a collection of a type "EMPLOYEE_TYPE", symbolic of an employee. The cursor for this container would then provide access to all the employees in the container. Additional constraints are usually allowed on cursor definitions to limit the elements referenced by the cursor. The example uses a department number field, "dept.no", to limit the employees referenced by the cursor to employees in department 10. See Figure 3.2.

```
// Declaration of the employee container.
container <EMPLOYEE_TYPE> employee;

// Cursor that references all elements in the employee container.
cursor <employee> all_employees;

// Cursor that references selected elements of employee container.
cursor <employee> where "$.deptno == 10" selected_employees;
```

FIGURE 3.2: Example Collection, Cursor and Selective Cursor Declarations

LEAPS makes use of composite containers. Composite containers model complex relationships across multiple containers. Consider three containers C_1, C_2, C_3. A relationship between the containers is the set of triples, where a triple is defined as (e_1,e_2,e_3), where e_i is an element of C_i.

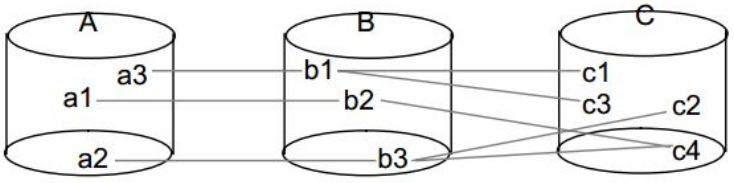

FIGURE 3.3: Example Composite Container Relationships

As an example Figure 3.3 illustrates the following triples formed from the three containers A, B and C.

$$(a3, b1, c1), (a3, b1, c3), (a1, b2, c4), (a2, b3, c2), (a2, b3, c4)$$

A composite cursor provides access to all n-tuples that can be produced from a relationship between containers. A composite cursor is then actually an n-tuple of cursors, one cursor per container. By advancing the composite cursor, successive n-tuples can be retrieved.

Selective composite cursors can also be declared. Figure 3.4 shows an example where a composite cursor **c** is defined that joins elements of the department and employee containers into 2-tuples if the "dept.no" field of both elements is equal.

```
compcurs < d department, e employee > where "$d.deptno == $e.deptno" c;
```

FIGURE 3.4: Example Composite Cursor Declaration

It is also possible to "seed" a composite cursor declaration. This is done when tuples that contain a particular element at a particular position are the only tuples needed. Using the example in Figure 3.3, it is possible to define a composite cursor that only returns triples that contain the element **b3** from container B, i.e. (a2,b3,c2) and (a2,b3,c4).

3.2.2.2 Algorithm

LEAPS maintains a Last In First Out (LIFO) stack of handles for working memory facts. Whenever a fact in working memory is inserted or deleted a time-stamp is associated with it. A handle for this insertion/deletion is then pushed onto the stack. LEAPS does not update elements, instead old versions are deleted and new versions are re-inserted. The reason for this is explained in a later section.

The handles in the stack are then iterated over one by one. When a handle is at the top of the stack, it is referred to as the dominant object (DO). The next step is the seeding of rule selection predicates. Relevant rules for checking are determined based on the predicate in the DO, i.e. only rules that have a condition element that contains the predicate are considered. This could be thought of as a form of condition membership. Rules may also be ordered by the number of positive condition elements that appear in the rule.

Essentially, a composite cursor is created with the DO as the seed. Each condition element of the rule corresponds directly to a container that is to be joined. In this case a container holds all the facts that match the type of the predicate in the relevant

condition element in the rule. It is important to note that the composite cursor has a restriction so only facts that have time-stamps less than or equal to the DO are considered. This is in the interest of fairness so that no n-tuple can fire a rule more than once.

An instantiation for the rule is then sought after by iterating through the composite cursor tuples. If the DO does not satisfy the relevant condition element of a rule i.e. no tuples are in the composite cursor, the next rule is considered.

When an instantiation is found, the current state of the search is stored on the stack and the resulting action is fired straight away. Processing then resumes with a new DO (which was added to the stack by the executed action) or the previous DO with the stored state. Once the DO has run out of rules to check, it is popped from the stack and the next handle becomes the new DO. Reasoning is completed when the stack is empty.

3.2.2.3 Deletion and Negative Rule Conditions

As discussed earlier, negative condition elements represent disqualification filters. The LEAPS interpretation of negation is depicted in Figure 3.5.

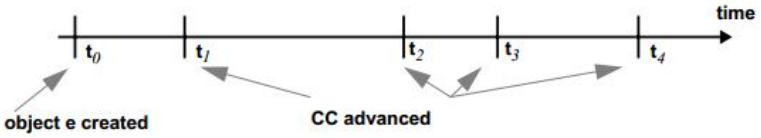

FIGURE 3.5: LEAPS Interpretation of Negation

A fact, e, is created at time t_0 and is used to seed an n-tuple by advancing a composite cursor at times $t_1 ... t_4$. Let P be the predicate of a negative condition element and t be the time of a composite cursor advancement. LEAPS determines if P is true at time t or at any time since e has been created.

For this approach to work, LEAPS must contain a history of facts so that time can be "rolled back" to evaluate P. Therefore, for every container C in the system there is also a "shadow" container S. When a fact is removed from C, it is inserted into S and a new handle is placed onto the stack. As stated previous, every fact has a time-stamp associated with it. When the fact is in container C the time-stamp indicates when the

fact was inserted, when the fact is in container S the time-stamp indicates when it was deleted. Elements in S are never modified.

For negated condition elements the following steps must be performed. Firstly, the container of the negative condition element is tested for any fact that satisfies predicate P of the negative condition element. Secondly, the corresponding shadow container is tested for any fact that satisfies P and whose time-stamp is greater than the dominant time-stamp. If either of the two steps return a fact that qualifies, the negative condition element fails.

It has been mentioned that handles for deleted facts are placed on the stack. It is clear what the intention of seeding of rule selection predicates with non-deleted facts is, however seeding with deleted facts is not immediately obvious.

The presence of a fact in a container may block the satisfaction of a negative condition element. With the deletion of the fact, previously blocked n-tuples may now be unblocked. As a result, when a deleted fact handle is at the top of the stack, the rule is modified for the search by changing the negative condition element to a positive condition element. The shadow container fact is then used to seed the resultant composite cursor.

The LEAPS approach can be characterized by lazy evaluation to avoid materialising tuples unnecessarily, by depth-first firing, and by the introduction of timestamps to set up temporal constraints, which can be used for handling deletion efficiently [27].

3.2.2.4 LEAPS Example

A brief example of the LEAPS algorithm is given to demonstrate the basic functionality. For simplicity, this example assumes the use of one rule which is shown below.

$$(?a \;\; rdf{:}type \;\; ?b),\ (?b \;\; rdfs{:}subClassOf \;\; ?c) -> (?a \;\; rdf{:}type \;\; ?c)$$

The example also makes use of the working memory facts in Table 3.1.

(a rdf:type C1)
(b rdf:type C1)
(C1 rdfs:subClassOf C2)
(c rdf:type C2)
(d rdf:type C3)
(C2 rdfs:subClassOf C3)
(C3 rdfs:subClassOf C4)

TABLE 3.1: LEAPS Example Working Memory Facts

The two condition elements will result in the creation of two containers. Initially these containers are empty. Once each working memory fact has been added, they become populated.

The stack is also initially empty. Each time a fact is added a new handle is pushed onto the stack. In this case the latest handle will relate to the last item added, (C3 rdfs:subClassOf C4) with the time-stamp 7. This is now the DO.

The current state of the containers, working memory and stack can been seen in Figure 3.6. Note the number to the left of the fact indicates the time-stamp associated with the fact and reflects the order in which it was added.

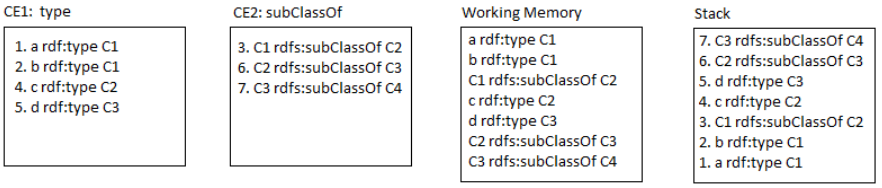

FIGURE 3.6: Populated Containers, Working Memory and Stack

The next step is to retrieve the relevant rules for this DO. As there is only one rule for this example, this rule will always be the rule considered.

Next, the composite cursor is created with the DO as the seed. The first cursor in the composite cursor is for the "type" container. This cursor will be pointing at the latest added fact to this container which is the (d rdf:type C3) fact with the time-stamp 5. The second cursor in the composite cursor will be fixed on (C3 rdfs:subClassOf C4) as this is the DO.

This first tuple happens to satisfy the rule and so this search is paused and the current state of the search is pushed onto the stack with the handle. The result of the rule firing is the addition of a new working memory fact, (d rdf:type C4). This also causes a new handle to be pushed onto the stack for this fact. See Figure 3.7.

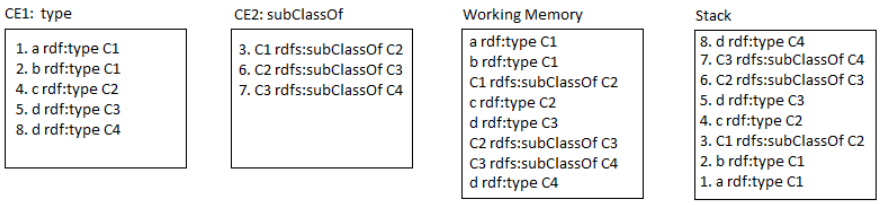

FIGURE 3.7: Updated Containers, Working Memory and Stack

This new handle will relate to the last item added, (d rdf:type C4) with the time-stamp 8. This is now the DO.

The next step is to create the composite cursor with the DO as the seed. In this case the first cursor in the composite cursor will be fixed on (d rdf:type C4) as this is the DO. The second cursor in the composite cursor is for the "subClassOf" container. This cursor will be pointing at the latest added fact to this container which is the (C3 rdfs:subClassOf C4) fact with the time-stamp 7.

This tuple does not satisfy the rule and so the composite cursor steps forward by moving the second cursor onto the next fact, (C2 rdfs:subClassOf C3) fact with the time-stamp 6. Again this does not satisfy the rule, and neither does the next fact with the time-stamp 3. This search is said to be exhausted, and so at this stage the next rule would be checked. As only the one rule is being considered in this example, this handle is popped from the stack and the next handle is considered.

The next handle is actually the first handle that was encountered, the handle that fired the action and was paused. The search resumes from the point where it stopped. (C3 rdfs:subClassOf C4) is again the DO, and the first cursor in the composite cursor now points at (c rdf:type C2) with the time-stamp 4. Recall that only facts with time-stamps lower than or equal to the DO are considered. This fact does not satisfy the rule and neither does the next two facts with time-stamps 2 and 1 respectively. This search is exhausted and so the handle is popped.

The next handle will relate to (C2 rdfs:subClassOf C3) with the time-stamp 6. This is now the DO.

Again, the next step is to create the composite cursor with the DO as the seed. The first cursor will be pointing at the first fact in the container with a time-stamp less than the DO, which is (d rdf:type C3) with the time-stamp 5. This does not satisfy the rule and so the next fact is considered, (c rdf:type C2) with the time-stamp 4.

This tuple happens to satisfy the rule and so this search is paused and the current state of the search is pushed onto the stack with the handle. The result of the rule firing is the addition of a new working memory fact, (c rdf:type C3). This also causes a new handle to be pushed onto the stack for this fact.

This process continues until no handles remain on the stack. The final view of the system can be seen in Figure 3.8.

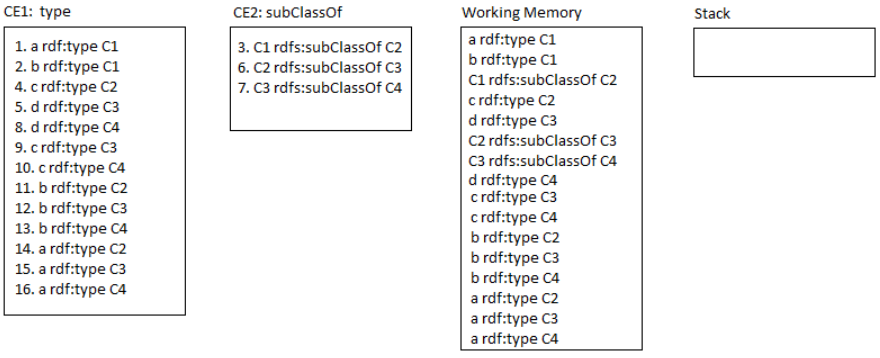

FIGURE 3.8: Final Containers, Working Memory and Stack

3.2.2.5 Comparison to RETE

The LEAPS algorithm offers the advantage of lazy evaluation. The RETE and TREAT algorithms are inherently slow, as they materialize all tokens that satisfy the predicate of a rule. These materialized tokens are stored in data structures and have a negative impact on performance as they must be updated as a result of executing rule actions.

A fundamental contribution of LEAPS is the lazy evaluation of tuples, i.e. the tuples are only materialised when needed. This approach drastically reduces both the space and time complexity of the reasoner and provides LEAPS with its phenomenal increase in rule execution efficiency [10].

3.2.3 Conclusion

The LEAPS algorithm can provide stronger performance guarantees than its predecessors, the TREAT and RETE algorithms, due to reduced asymptotic algorithmic complexity [28]. In particular, LEAPS requires only linear space versus exponential space for TREAT and RETE [25]. These properties are very important when considering the algorithm for use in a resource constrained environment.

While LEAPS may be significantly more difficult to implement compared to TREAT, it too can also retain the operational semantics defined by the RETE algorithm. This suggests that once implemented it can be integrated into SCOROR without much difficulty.

The reasons above lead to the decision to choose LEAPS as the algorithm to be implemented in this research.

3.3 Summary

This chapter has given an in depth discussion on two reasoning algorithms, TREAT and LEAPS. Both algorithms were discussed in detail, with advantages and disadvantages for both approaches in relation to RETE given.

The discussion lead to the conclusion that LEAPS could offer a more interesting replacement for RETE in SCOROR due to its stronger performance guarantees.

The next chapter will discuss the design of the LEAPS algorithm so that it may be integrated into SCOROR.

Chapter 4

Design

4.1 Introduction

This chapter gives a high level design of the LEAPS algorithm. The aim of the chapter is to provide the reader with a design which illustrates the key concepts of the approach and that may enable the implementation of the LEAPS algorithm in a chosen language.

At this stage it is important to note that the rules used in SCOROR and the research performed by Hardy did not support the use of negative condition elements. For this reason, and in the interest of comparing the implementation of the RETE algorithm to the chosen algorithm accurately, negative condition support was not included in this research implementation of LEAPS.

4.2 LEAPS

The LEAPS algorithm and the concepts associated with it have been described in the State Of The Art chapter. The basic outline of the intended implementation will be given in this section, with specific implementation details given in the Implementation chapter.

The design of the LEAPS algorithm implementation is taken from DROOLS [29]. DROOLS is a business rule management system (BRMS) with a forward chaining inference based rules engine. The project is open-source and offers a reasoning system which currently uses the RETE algorithm.

The complete stream reasoning process is as follows: initial rule loading, insertion of static ontology triples, reasoning cycle, insertion of current triples in stream window, reasoning cycle, sweeping of reasoner. The final three steps are repeated continuously until the stream or the reasoning process is stopped.

4.2.1 Rule Loading

The first phase of the reasoning process is to load the rules into the reasoner that will be used for reasoning. This process is currently already performed by COROR and is independent of what reasoning algorithm is being used. This is as simple as parsing well defined rules from a text file. As an aside, the required rule composition is given below. The rules used in this research are given in Appendix A.

4.2.1.1 Rule Composition

An example of a rule is given below.

$$(?a \;\; rdf{:}type \;\; ?b),\ (?b \;\; rdfs{:}subClassOf \;\; ?c) -> (?a \;\; rdf{:}type \;\; ?c)$$

Rules consist of both condition elements and actions. Any number of both components is allowed, as long as each pattern is enclosed within brackets and each pattern is comma separated. The condition elements should be separated from the actions with a simple arrow pattern, composed of a hyphen and a greater than symbol.

It has been mentioned that COROR supports a selective rule loading mechanism. This algorithm examines the base ontology and determines what rules will be needed in

the reasoner. This can help reduce the overall reasoning time as less rules need to be examined. Unfortunately, this is not suitable for a streaming environment as there is no way to know what sort of triples may arrive on the stream.

Although COROR performs the initial rule loading, there is a conversion process in place that converts the rule into a format which is used by the LEAPS approach. This must be performed to examine if there are any variables in place within the rule condition elements. If there are, variable bindings and relevant conditions must be set up for the rule. Take the following rule for example:

$$(?a \;\; rdf{:}type \;\; ?b),\ (?b \;\; rdfs{:}subClassOf \;\; ?c) -> (?a \;\; rdf{:}type \;\; ?c)$$

In this case, two triples could be found that satisfy the two condition elements independently. However, a third condition would need to be in place to ensure that both the object of the first condition element and the subject of the second condition element have the same value.

For example, the triples in Table 4.1 may exist in working memory.

(C2 rdfs:subClassOf C3)
(C1 rdfs:subClassOf C2)
(a rdf:type C1)

TABLE 4.1: Example Working Memory Facts

If the facts are inserted into the reasoner in descending order from how they appear in the table, the final fact, who's predicate is "type", will be considered as the DO. The relevant rules will return the rule given above, and the DO will seed the search by satisfying the first condition element *(?a rdf:type ?b)*.

The two facts considered for the second condition element will be the facts with the predicate "subClassOf". The newest fact, *(C2 rdfs:subClassOf C3)*, will be considered first. Clearly this fact does satisfy the second condition element as the predicates match and the other values are variables. However, the values given for the b variable by the two facts will differ. For this reason an additional check to ensure matching variable bindings for different condition elements must be performed. This can be done by

maintaining a record of each variable used in the rule and how many times this variable is referenced. When a fact is considered to satisfy a condition element, any variables satisfied should store the relevant value so that this variable reference can be checked against other variable references in the rule. The final variable value may also be used in the rule's actions.

The actions of the rule must also be expressed in a way that makes sense in the LEAPS environment. In this case, an action would need to be in place that inserts a new triple with the values that have been bound to the a and c variables. Using the above example, this would be the triple below.

$$(a \quad rdf{:}type \quad C2)$$

The rule loading process is also important due to the fact that it is responsible for setting up the data structures which are used in the reasoning process. Once reasoning begins, the first step in the algorithm is to use the DO predicate to determine what rules are needed for checking. This information is gathered at this stage by examining the predicates in each rule condition element as each rule is loaded.

The rule loading process should construct a collection of rules for each unique predicate. This collection should hold all the rules that reference that particular predicate, and should indicate which condition element referenced the predicate. When a fact is then determined as the DO, relevant rules can be retrieved from the collection.

For example, consider the rules in Table 4.2.

(?a rdf:type ?b), (?b rdfs:subClassOf ?c) -> (?a rdf:type ?c)
(?a rdf:type rdf:Property) -> (?a rdfs:subPropertyOf ?a)
(?a owl:equivalentClass ?b) -> (?a rdfs:subClassOf ?b)

Table 4.2: Example Rules

As each rule is loaded, a new collection is created for each unique condition element predicate that is encountered. The collections are illustrated in Figure 4.1. The relevant condition element is highlighted.

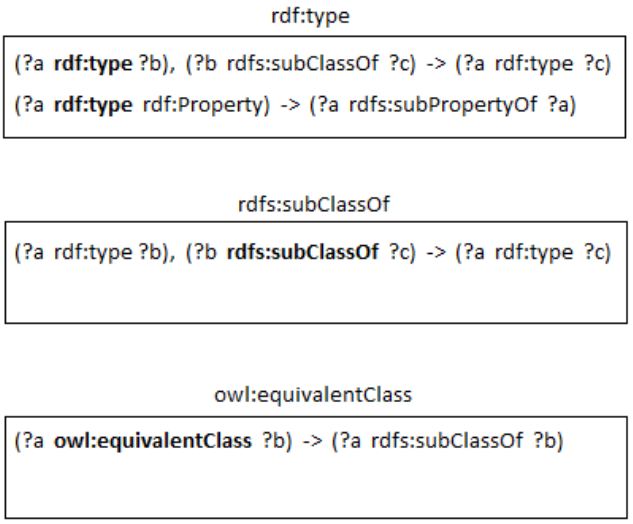

FIGURE 4.1: Rule Containers

A wild-card flag is also used to indicate if a wild-card predicate is encountered, that is to say a rule where the predicate is a variable. If a single rule contains a wild-card predicate, this flag should be set. This flag is later used in the triple insertion stage.

4.2.2 Insertion of Ontology Triples

The next step in the reasoning process is the insertion of the triples contained in the static ontology. COROR takes full advantage of information gathered in the rule loading stage for further efficiency.

There are two possible cases. The first case is that there is at least one rule that contains a wild-card predicate. The second case is that all predicates that appear in rules are known. In the first case the wild-card flag will be set and the reasoner must insert all triples from the ontology, as it is possible for the wild-card predicate to be satisfied by any of the triples. However, in the second case the reasoner must only insert triples that contain predicates that have been already observed in the rules.

For example, consider the rules from Figure 4.1 have been loaded. In this case, the wild-card flag would not be set as there are no wild-card predicates used in any of the condition elements. Only facts that contain the predicates that have been referenced

would need to be inserted. If the facts from Table 4.3 were to be inserted, the triples in bold could safely be ignored, as they could not possibly satisfy any condition elements.

(b rdfs:domain C1)
(C2 rdfs:subClassOf C3)
(C1 rdfs:subClassOf C2)
(a rdf:type C1)
(d rdfs:range C3)
(c rdf:type C3)
(C2 owl:equivalentClass C4)
(e rdf:type C4)

TABLE 4.3: Further Working Memory Facts

This simple step avoids the insertion of triples which, when used as the DO, would have no rules to consider. These triples would consume space and time during reasoning.

The insertion process also involves the creation of the collections which hold all facts for a certain predicate type. These collections are the containers that allow for the creation of composite cursors, discussed in Section 3.2.2.1, whose use is described in Section 3.2.2.2.

As each fact is inserted, a new collection is created for each unique predicate that is encountered. The collections generated for the facts in Table 4.3 are illustrated in Figure 4.2.

As each triple is added into the reasoner, a handle for the triple is also placed onto the LIFO stack. The stack for the facts in Table 4.3 is illustrated in Figure 4.3.

4.2.3 Reasoning

Reasoning then proceeds as discussed in Section 3.2.2.2. Each handle on the stack is considered in this process. When a handle is at the top of the stack it is referred to as the DO. The relevant rules can now be retrieved by using the predicate in the fact the handle represents and the rule containers which have been created in the rule loading stage. The DO is then used as a seed for the composite cursor. The composite cursor

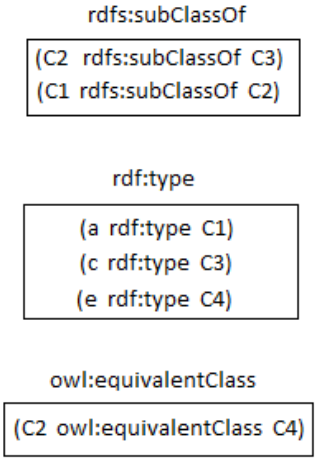

FIGURE 4.2: Fact Containers

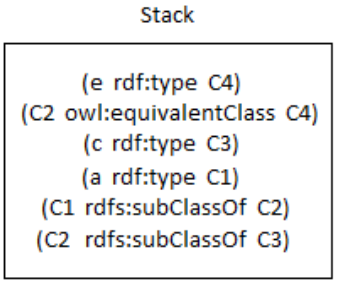

FIGURE 4.3: LIFO Stack

consists of N cursors, where N is the number of condition elements in the rule (the cursor for the DO is fixed on the DO fact). Each of the other cursors then iterates over the relevant container of facts for their respective condition element predicate. These fact containers were created in the triple insertion stage.

For example, consider the stack in Figure 4.3, the rule containers in Figure 4.1 and the fact containers in Figure 4.2.

The first DO is then the *(e rdf:type C4)* fact. The relevant rules to check can then be retrieved from the "rdf:type" container. The first cursor will be fixed on the DO fact, the second cursor will iterate over the "rdf:subClassOf" fact container to attempt to find an instantiation. If none is found, the fact container cursor is reset and the next rule in the rule container is considered. When no rules are left, the handle is popped from the stack and the next handle becomes the DO.

When an instantiation is found the handle is pushed back onto the stack, and the current iteration state of the composite cursor (i.e. the positions of the individual cursors) is retained within it. The rule's action is then executed. New insertions result in the pushing of new handles onto the stack.

When a handle reaches the top of the stack and an iteration state already exists, the search continues from the saved state.

Once initial reasoning over the base ontology has finished, stream reasoning may begin. Any triples inserted and deduced will now be considered temporal triples. When temporal triples are inserted they are still added to the relevant fact container. As discussed in Section 2.4 these temporal triples must be removed once their associated time-stamp is deemed invalid. This is explained in further detail in the next section.

4.2.4 Temporal Triple Requirements

When expanding COROR to support stream reasoning Hardy [14], based on the approach by Barbieri et al [23], identified 7 main requirements for a stream based RETE reasoner. The requirements are listed below.

1. Computes the entailments derived by the inserts.
2. Annotates each entailed triple with an expiration time.
3. Eliminates from the current state all copies of derived triples except the one which has the highest time-stamp.
4. All tokens placed in RETE network must be removed at expiration time.
5. Products of joins between two temporal tokens must output a temporal token with the minimum time-stamp of its two parents. The product of a triple with a temporal triple must output a product with time-stamp of its temporal parent.
6. Facts contained in the base ontology should not be replaced by temporal versions of themselves.

7. Tokens, similar to triples in the ontology, must be updated in the case of duplicate tokens arriving into the RETE network if we are to preserve correctness in our joins.

The requirements are highly relevant for the design of the LEAPS reasoner. The requirements, as well as the explanations, are presented below to reflect the needs of the LEAPS approach.

1. **Computes the entailments derived by the inserts.**

   The first requirement is a basic reasoning requirement, the reasoner must be able to take a set of facts and be able to deduce the appropriate entailments.

2. **Annotates each entailed triple with an expiration time.**

   By labelling entailments with the minimum time-stamp of the facts used to deduce them, triples can be safely removed from the graph based on their time-stamps without having to worry about figuring out their consequences and seeing if they should be removed.

3. **Eliminates from the current state all copies of derived triples except the one which has the highest time-stamp.**

   This prevents the reasoner from having multiple duplicates of triples with different timestamps. The absence of this requirement would lead to large amounts of unnecessary computation and memory usage.

4. **All temporal handles placed on the LEAPS stack must be removed at expiration time.**

   Once a temporal triple has been removed from the reasoner, it is only logical that its associated handle is removed.

5. **Triples that have been deduced with the use of temporal triples must receive the minimum time-stamp of all temporal triples involved.**

   This is an intuitive step that is needed for deductions that are joined from multiple facts that may expire. If there is a conclusion Z that has been derived from facts X and Y, then it is necessary that Z's expiration time should be the minimum of

X and Y, as once one of the facts that is necessary for Z to exist expires, then Z should expire as well.

6. **Facts contained in the base ontology should not be replaced by temporal versions of themselves.**

   If a fact does not have a time-stamp originally then it is persistent data and should not be removed. This can lead to significant performance gains because if a base ontology fact is allowed to be changed to a temporal fact, then all of the conclusions that were derived from it will also become temporal. This can sometimes lead to large amounts of an ontology becoming temporal, which can greatly increase the overhead of the reasoning process.

7. **Handles, similar to triples in the ontology, must be updated in the case of duplicate handles arriving into the reasoner.**

   If a temporal triple is replaced by a duplicate temporal triple with a higher time-stamp, the relevant handle must be removed from the stack and pushed onto it again.

Thus restrictions are in place when inserting new triples that have been deduced using temporal triples. The triple must have the minimum time-stamp of all temporal triples that have been used in its deduction. If the triple is already in the working memory, and is not a temporal triple, it is not inserted. If the existing triple is a temporal triple, a check must be performed so that only the triple with the highest time-stamp is permitted to exist.

An extra step is also added to the reasoning process. This step is referred to as the "sweep" stage. During this step, the working memory, fact containers and stack are "swept" over. Any expired data is then removed from the reasoner. This step is performed after each reasoning cycle.

4.3 Summary

This chapter has given a high level design of the LEAPS algorithm. The significant concepts of the approach have been discussed in detail with examples given to illustrate

the approach.

The next chapter will explain how this design was implemented in Java and integrated into SCOROR.

Chapter 5

Implementation

5.1 Introduction

The existing stream reasoner, SCOROR, was implemented in Java. Thus, the LEAPS approach was implemented in Java. This chapter will discuss SCOROR at two levels, Graph Level Implementation and LEAPS Level Implementation.

The Graph level is independent of which reasoning algorithm is in use. The Graph handles how triples are managed in working memory, independent of the actual reasoner. When a triple is added to working memory it is first added to the Graph, before being passed into the reasoner. Any deduced triples are then passed back to the Graph. The Graph level provides a means to illustrate clearly the initial ontology size and the resulting deduced Graph size. Changes were made by Hardy [14] at Graph level to support stream reasoning. These changes will be discussed here.

LEAPS level relates to the specific implementation of the LEAPS approach. This involves the construction of data structures and also the search algorithms that are used in reasoning. Modifications to the LEAPS level that were needed to support stream reasoning are also discussed.

5.2 Graph Level Implementation

COROR makes use of multiple graphs during its operation. When triples are first inserted they are inserted into an instance of the *RawGraph* class, which is used to represent the base ontology. Deduced triples are inserted into an instance of *DeductionsGraph*. These are both members of *InfGraph*, which models a complete cycle of inference containing both the initial triple set and the result set.

Hardy added a new graph to *InfGraph* called *fadd*. The purpose of this graph is to hold all triples from the current stream window. These triples are added to the graph from the stream processor. Originally COROR only had the ability to reason over a static ontology. There was no functionality which allowed for the insertion of triples into a reasoner which had already reasoned over the static ontology. Subsequent calls on the reasoner would result in the complete process being performed again. This includes the rule loading, the construction of the RETE network and the static ontology insertion.

With the addition of the *fadd* graph and some simple logic, SCOROR was given the ability to perform incremental reasoning. On the initial call to the reasoner the rules are loaded and the relevant reasoner data structures are constructed. The base ontology (*RawGraph*) is then passed into the reasoner and normal reasoning is performed.

On subsequent calls, there is no need to reconstruct data structures. Triples which have been extracted from the current stream window (*fadd*) are then inserted into the reasoner as temporal triples. Upon completion of reasoning, the "sweep" call is performed. This call removes any triples from both the *RawGraph* and the *DeductionsGraph* whose times-stamps are below the current time. This cycle can then be repeated indefinitely.

To support temporal triples, Hardy extended the simple Triple class into the TemporalTriple class. This new class simply includes a time-stamp. To prevent duplicate temporal triples a simple test is performed upon insertion into a graph. The test checks if the graph already contains the triple. If it does not, the triple is added. If it does but the contained triple is a temporal triple, the time-stamps are compared. If the contained triple has a higher time-stamp, it is left in the graph. If not, it is removed and the new triple is inserted. This was implemented into the base graph class *graphImpl* to ensure that all graphs enforce the behaviour.

5.3 LEAPS Level Implementation

The core functionality of the LEAPS implementation is contained within the *LeapsWorkingMemory* and *LeapsRule* classes. These classes are integrated with the existing SCOROR project through the *LeapsEngine* class.

These classes can be seen in Figure 5.1.

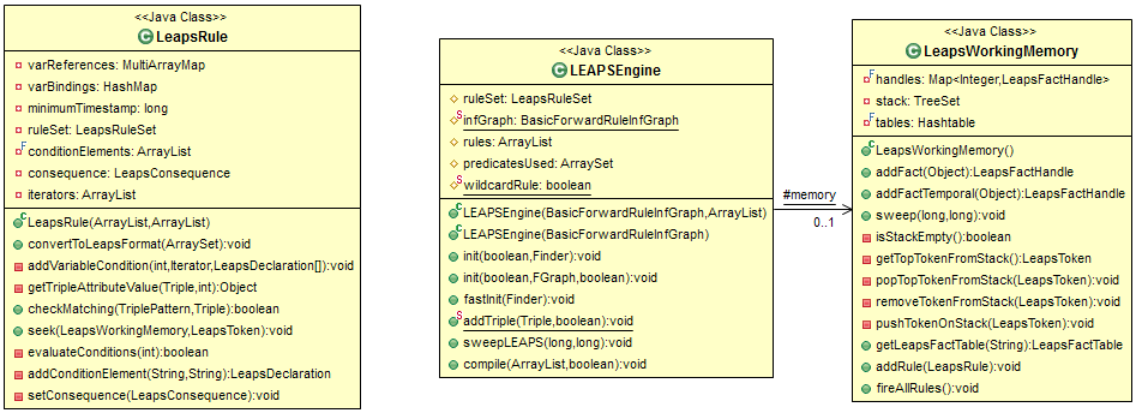

FIGURE 5.1: LeapsRule, LeapsEngine and LeapsWorkingMemory Classes

When SCOROR is started, it must first reason over the static base ontology before stream reasoning can begin. Firstly, SCOROR initialises the *LeapsEngine* class and passes in the rules which have been read in from the rule text file. A call to the "init" method is then made, which passes in the base ontology facts to be inserted, which have also been read in from a text file. A boolean parameter indicates that this is the first time "init" has been called and so the "compile" method is then called to perform further rule loading and generate the rule containers discussed in the Design chapter. The "fastInit" method is called to insert the base ontology facts into the reasoner, and generate the fact containers also discussed in the Design chapter. Finally, the "fireAllRules" method of the *LeapsWorkingMemory* class is called which initiates the reasoning process.

Once this reasoning has finished, further calls to "init" will bypass the "compile" method as the data structures needed will already have been generated on the first call. The facts which are passed into the method on these calls represent the facts from the current stream window. The "fastInit" method will be called with these facts, followed by the "fireAllRules" method. Once reasoning has complete, the "sweepLeaps" method is called in order to remove any invalid temporal triples and associated data.

Each of the above steps will now be discussed in detail.

5.3.1 Rule Loading

In COROR a rule is modelled with the *Rule* class. This was extended into the *LeapsRule* class. When the "compile" method from the *LeapsEngine* is called, the rules which have been passed to the method are iterated over and converted into the LEAPS format using the method "convertToLeapsFormat" of the *LeapsRule* class. This rule is then added to the rule set in the *LeapsEngine*.

When a rule is converted it must be examined in order for variable bindings, conditions and actions to be put in place. The *Rule* class splits the rule into two components: the body, which is made up of the condition elements, and the head, which is made up of the actions. Each condition element in the body is given an index, as is each action in the head. Each condition element and action must be an instance of the *TriplePattern* class, and this is enforced when the rules are loaded. The *TriplePattern* class ensures that each instance has a subject, predicate and object, each represented by an instance of a *Node* class. The *Rule* class also assigns an index to each variable in the rule.

For example, examine the following rule:

$$(?a \;\; rdf{:}type \;\; ?b),\ (?b \;\; rdfs{:}subClassOf \;\; ?c) -> (?a \;\; rdf{:}type \;\; ?c)$$

The body would consist of two elements: *(?a rdf:type ?b)* and *(?b rdf:subClassOf ?c)*. These elements would have the indices 0 and 1 respectively. The head would consist of the single element: *(?a rdf:type ?c)*, index of 0. The variables would be assigned the following indices: a=0, b=1 and c=2.

Also associated with each condition element in the rule is an iterator. When a search for an instantiation is being performed each iterator gets set so that it may iterate over a fact container. This fact container is usually the fact container for the predicate contained in the condition element. This will be discussed in more detail later.

For each condition element in the body, the "addConditionElement" of the *LeapsRule* is called. The operations performed in this method are described below.

5.3.1.1 Variable References

Each condition element is checked for variables in the subject, predicate and object positions. If a variable is found, a reference is stored in a collection called *varReferences* in the *LeapsRule* instance. The key for the reference is the variable index, while the value is an instance of the *Pair* class (see Figure 5.5) which contains the index of the condition element and the attribute the variable is contained in, i.e. subject, predicate or object. Obviously, a variable can be referenced multiple times in a rule so a key may return multiple pairs. The above rule would result in the references in Table 5.1 being stored.

Var Index	Pair: (CE Index, Attribute)
0	(0, SUBJECT)
1	(0, OBJECT)
1	(1, SUBJECT)
2	(1, OBJECT)

TABLE 5.1: Example Variable References

If a predicate is a variable, the wild-card flag in the *LeapsEngine* is set. Otherwise, the predicate value is added to a *predicatesUsed* collection contained in the *LeapsEngine*. This information is later used in the triple insertion process as discussed in Section 4.2.2.

5.3.1.2 Conditions

The next step is to enforce the actual condition element. For each condition element a *LeapsCondition* is registered. These conditions are then tested during reasoning time. Each *LeapsCondition* contains an "isAllowed" method that must ensure that the current triple pointed to by the condition element's iterator passes the condition element by matching the pattern of the condition element.

A triple matches a pattern by matching the value for each of the three positions: subject, predicate and object. If any of the three is a variable, any value may pass as a match. This value is stored in another collection in the *LeapsRule* instance, *varBindings*, so that the value may be used in the rule's actions.

If the triple is a temporal triple, its time-stamp is stored (if it is less than a previously stored time-stamp) so that it may be used by the actions for the insertion of a new triple.

Pseudo code of this operation is given in Figure 5.2 and Figure 5.3. The "isAllowed" method makes use of the "checkMatching" method in the *LeapsRule* instance.

```
boolean isAllowed(){

    TriplePattern pattern = this.pattern;
    Triple trip = iterator.getValue();


    boolean test = checkMatching(pattern,trip);

    if(test && trip instanceof TemporalTriple){

        if(this.minTimestamp > ((TemporalTriple) trip).getTime()){
            this.minTimestamp = ((TemporalTriple) trip).getTime();
        }
    }
    return test;
}
```

FIGURE 5.2: "isAllowed" Pseudo Code

```
boolean checkMatching(TriplePattern pattern, Triple fact){
    Node subject = pattern.getSubject();
    Node predicate = pattern.getPredicate();
    Node object = pattern.getObject();

    boolean s = true,p = true,o = true;

    if(subject.isVariable()){
        varBindings.put(getVariableIndex(subject), fact.getSubject()) ;
    }else{
        s = fact.getSubject().equals(pattern.getSubject());
    }
    if(!s){return false;}

    if(predicate.isVariable()){
        varBindings.put(getVariableIndex(predicate), fact.getPredicate()) ;
    }else{
        p = fact.getPredicate().equals(pattern.getPredicate());
    }
    if(!p){return false;}

    if(object.isVariable()){
        varBindings.put(getVariableIndex(object), fact.getObject()) ;
    }else{
        o =  fact.getObject().equals(pattern.getObject());
    }
    if(!o){return false;}

    return s && p && o;
}
```

FIGURE 5.3: "checkMatching" Pseudo Code

5.3.1.3 Variable Bindings

An extra condition is also added for each variable in the rule that has more than one reference. This is determined using the *varReferences* information described above. If a variable has more than one reference an additional *LeapsCondition* is put in place using the "addVariableCondition" method of the *LeapsRule* instance.

This *LeapsCondition* makes use of the information in *varReferences* to determine all of the condition elements that reference the variable in question. Recall that the information also indicates what attribute the variable is in in the condition element i.e. subject, predicate or object.

Using this information, the triples pointed to by each condition element's iterator can be obtained and checked against each other. This ensures that all references to a variable contain the same value. If this is so, the condition passes and this variable's value is updated in the *varBindings* collection.

Pseudo code for this operation is given in Figure 5.4 and Figure 5.5. The "isAllowed" method makes use of the "getTripleAttributeValue" method in the *LeapsRule* instance and the *Pair* class, which are also shown.

```
Let:
    varIndex = index of the variable in question.


boolean isAllowed(){

    boolean ret = true;
    Iterator it =  varReferences.getAll(varIndex);

    Pair pair = (Pair) it.next();
    Object last = getTripleAttributeValue(iterators[pair.ceIndex].getValue(),
                                          pair.atrributeIndex);

    while(it.hasNext() && ret){
        pair = (Pair) it.next();
        Object current = getTripleAttributeValue(iterators[pair.ceIndex].getValue(),
                                                 pair.atrributeIndex);
        ret &= last.equals(current);
        last = current;
    }
    if(ret){
        varBindings.put(varIndex, last);
    }

    return ret;
}
```

FIGURE 5.4: "isAllowed" Pseudo Code

5.3.1.4 Actions

The next step in rule loading is to set the actions of the rule. The rules which are used by COROR do not include removal of triples as actions. All actions result in the insertion of one triple, however the LEAPS implementation can support any number of insertions. This is done by setting a *LeapsConsequence*. Every consequence has an "invoke" method which is executed when the rule is satisfied.

```
Triple getTripleAttributeValue(Triple triple, int index){
    switch(index){
        case 0:
            return triple.getSubject();
        case 1:
            return triple.getPredicate();
        case 2:
            return triple.getObject();
        default:
            return null;
    }
}



class Pair {

    int ceIndex; //Condition Element index in the Rule
    int atrributeIndex; // 0,1,2 = Subject,Predicate,Object

    Pair(int ceIndex, int atrributeIndex){
        this.ceIndex=ceIndex;
        this.atrributeIndex = atrributeIndex;
    }
}
```

FIGURE 5.5: "getTripleAttributeValue" and *Pair* class Pseudo Code

The method simply iterates through the head and performs an insertion with the values specified in the relevant action. If a variable is present in the action the value for the variable can be retrieved from *varBindings*.

Pseudo code for this operation is given in Figure 5.6.

```
void invoke(){
    for (int i = 0; i < headLength(); i++) {
        Object clause = getHeadElement(i);
        TriplePattern pattern = null;

        if (clause instanceof TriplePattern) {
            pattern = (TriplePattern) clause;
            Node subject = pattern.getSubject();
            Node predicate = pattern.getPredicate();
            Node object = pattern.getObject();

            if(subject.isVariable()){
                subject = (Node) varBindings.get(getVariableIndex(subject));
            }
            if(predicate.isVariable()){
                predicate = (Node) varBindings.get(getVariableIndex(predicate));
            }
            if(object.isVariable()){
                object = (Node) varBindings.get(getVariableIndex(object));
            }

            if(minTimestamp < Long.MAX_VALUE){
                //minimumTimestamp has been changed
                TemporalTriple t = new TemporalTriple(subject,predicate,object, minTimestamp);
                LEAPSEngine.addTriple(t,true);
            }
            else{
                Triple t = new Triple(subject,predicate,object);
                LEAPSEngine.addTriple(t,true);
            }
        }
    }
}
```

FIGURE 5.6: "invoke" Pseudo Code

5.3.1.5 Creation of RuleTables

Finally, the rule is added to the rule set. At this stage it is important to explain some of the more complex data structures which are used in this implementation. One such structure which is heavily used is the *LeapsTable*. This is a custom data structure which is based on a TreeMap. It allows for the addition and removal of objects which are always maintained in a sequential order. This order is based on the order in which objects are inserted, but can be given a comparator parameter when created which defines how objects are sorted. For example, this could allow for the rules to be sorted by some measure of complexity.

The *LeapsTable* structure also offers iterators which allow for iteration over the objects contained in the structure. These iterators can be given parameters which allow for the iterator to start and end at particular objects. It should be evident that the *LeapsTable* structure is an implementation of the Container concept introduced in Section 3.2.2.1.

In fact, a structure called a *FactTable* extends *LeapsTable* and is used to store fact-handles for all the facts for a certain predicate type. A structure called a *RuleTable*, which also extends *LeapsTable*, is used to store rule-handles for all the rules that reference a particular predicate type. Each *RuleTable* instance is actually contained within the relevant *FactTable* instance. The *FactTable* then provides methods to add a rule to the *RuleTable*, and to fetch the next rule-handle.

These classes can be seen in Figure 5.7.

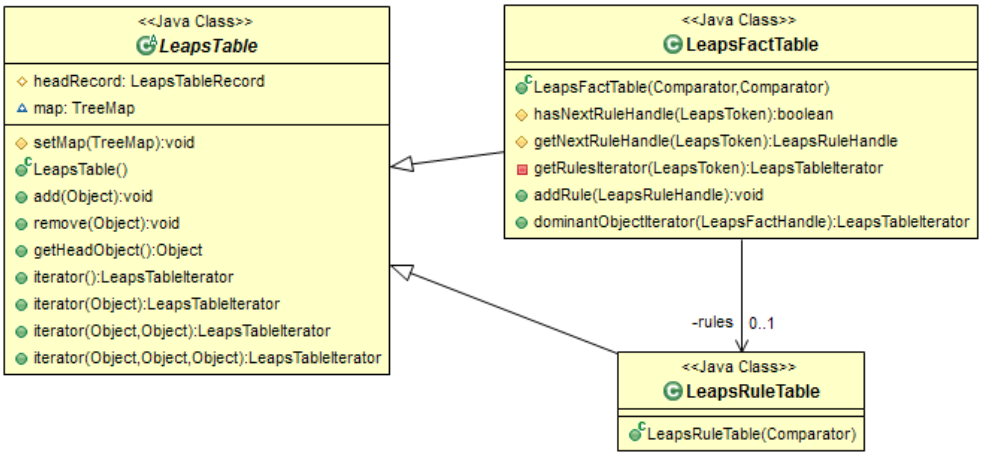

FIGURE 5.7: LeapsTable, LeapsFactTable and LeapsRuleTable Classes

With these structures explained, it is possible to return to the rule loading process. As each rule is added to the rule set a new *FactTable* for each unique condition element predicate is created and stored in the *tables* collection in the *LeapsWorkingMemory* class. When all rules have been loaded, there will exist a *FactTable* and *RuleTable* for each unique predicate. This *RuleTable* table will contain a rule-handle to all the rules that reference this particular predicate. Each rule-handle will also contain the index of the condition element the predicate was referenced in. The rule-handle must hold this information so that when it is used in the reasoning process, the DO will know which condition element to seed.

For example, consider that the following rule has been loaded.

$$(?a \ \ rdf{:}type \ \ ?b),\ (?b \ \ rdfs{:}subClassOf \ \ ?c),\ (?c \ \ rdfs{:}subClassOf \ \ rdfs{:}Resource)$$

$$-> \ (?a \ \ rdf{:}type \ \ ?c)$$

Firstly, a new *FactTable*, and subsequently a new *RuleTable*, for the predicate "rdf:type" would be created. A rule-handle would then be added to the *RuleTable* using the "addRule" method. The rule-handle would contain a reference to the *LeapsRule* object itself, and the index 0.

Secondly, a new *FactTable*, and subsequently a new *RuleTable*, for the predicate "rdfs:subClassOf" would be created. A rule-handle would then be added to the *RuleTable* using the "addRule" method. The rule-handle would contain a reference to the *LeapsRule* object itself, and the index 1.

Lastly, a rule-handle would be added to the *RuleTable* for the predicate "rdfs:subClassOf". The rule-handle would contain a reference to the *LeapsRule* object itself, and the index 2.

It is important to note that a *RuleTable* may exist for the wild-card predicate.

5.3.2 Insertion of Ontology Triples

When the "fastInit" method is called, facts are inserted at the graph level and also at reasoner level. The "addTriple" method in the *LeapsEngine* class is called, which in turn calls the "addFact" method of the *LeapsWorkingMemory* class.

When facts are inserted a new *FactTable* for each unique predicate is created unless it already exists from the rule loading stage. This table will contain a fact-handle to all the facts that contain this predicate. A fact-handle is a simple object that contains a reference to the relevant *Triple* object.

A new *LeapsToken* is created for each new fact-handle. A token maintains a reference to the fact-handle that created it. It also contains a resume flag which indicates whether its fact has been considered as the DO yet or not. It also contains a reference to the last rule-handle used in the search with the token, and stores the saved iteration state of the search.

This token is pushed onto the stack. The stack is implemented as a TreeSet which orders inputs based on their insertion. A TreeSet was chosen over a standard Stack class as it can be faster for removing tokens for retracted facts. Although COROR does not support rule actions that remove triples, temporal triple removal is needed for stream reasoning.

5.3.3 Reasoning

5.3.3.1 Reasoning Cycle

When a token is at the top of the stack, the relevant *FactTable* is retrieved from *tables* using the predicate from the fact. The resume flag states whether it is resuming a search or if this is the first time the fact is being used as the DO. If the flag is set then the search continues from the saved iteration state which is stored in the token.

The first time a token reaches the top of the stack the first rule-handle is retrieved from the *LeapsRuleTable* and is set as the current rule-handle for the token. The "Seek" method is then called for the rule referenced by the rule-handle. If an instantiation for the rule is found, the iteration state is saved onto the token. The resume flag for the

token is then set. The resulting action will have inserted a new fact, so the new token on the top of the stack is now considered as the DO.

However, if no instantiation is found, the next rule-handle is retrieved from the *LeapsRuleTable*. When there are no more rule-handles available from the *LeapsRuleTable*, a new *LeapsRuleTable* may be used which represents rules that contain wild-card predicates. This depends on whether the wild-card flag is set or not in the rule loading stage. When all possible rule-handles have been considered, the token is popped from the stack. This process repeats until there are no tokens left on the stack.

5.3.3.2 Seek Method

The Seek method is used to find instantiations for a rule which has been seeded with a DO. The method is called for a particular rule when a rule-handle which is associated with the rule is chosen in the reasoning cycle. As mentioned earlier, each rule-handle also contains the index of the condition element the predicate was referenced in. This condition element is the element that is seeded.

It was also mentioned earlier that each condition element in a rule has an associated iterator. The first step in the method is to set these iterators so that they iterate over the relevant *LeapsFactTable* instance so that facts that satisfy the condition elements can be found. Obviously, the iterator for the condition element that is seeded by the DO is restricted to just the DO fact. This iterator is retrieved using the "dominantObjectIterator" method from the *LeapsFactTable* class.

If the search is resuming the iterators can be set back to the state they were in at the end of the last search using the saved iteration state in the token. If the saved iteration state points to a fact that has been removed from a *LeapsFactTable* since the last search, the iterator may need to be reset to prevent these facts from being considered. Once the iterators have been set, a simple check can be performed to ensure there are no empty iterators. If there is, the method can return reporting that no instantiation is found.

The next step is to cycle through the iterators attempting to find facts that satisfy the condition elements. Starting with the first iterator, the relevant *LeapsConditions'* "isAllowed" methods are called using the "evaluateConditions" method for the *LeapsRule*.

If the method returns that the conditions have passed, the search can proceed to the next iterator. Once the last iterator has found a fact that satisfies the conditions, the current iteration state of the iterators is saved in the token, the consequence is invoked using the "invoke" method and the "Seek" method returns stating an instantiation has been found.

If an iterator fails to find a fact to satisfy it and has no more facts to check, it is reset and the algorithm moves back to the previous iterator which moves onto the next fact. If the first iterator contains no more facts to check, the search can return reporting that no instantiation has been found.

5.3.4 Temporal Triple Management

In order to support stream reasoning, some additional functionality had to be employed in order to support the insertion and removal of temporal triples.

When a temporal triple is inserted, a check is done to assess whether the triple is already contained in the *InfGraph*. Inserting a duplicate triple into the graph will result in the duplicate being handled as discussed in Section 5.2. When inserting a temporal triple in the reasoner, a search of the *InfGraph* is first performed in the "addTriple" method in the *LeapsEngine* class. If the triple already exits, the "addFactTemporal" method of the *LeapsWorkingMemory* class is then called.

The existing triple's fact-handle can be retrieved using a hashing technique and the new triple object. All fact-handle objects must be stored with their corresponding hash for this approach. If the existing triple is not a temporal triple, no action is performed. However, if the existing triple is a temporal triple the time-stamps must be compared. If the existing triple has a higher time-stamp, the existing fact-handle is left in the *LeapsFactTable* and the respective token in the stack. If not, it must be removed from the *LeapsFactTable* and the token must be removed from the stack. The new temporal triple's fact-handle is then re-inserted into the relevant *LeapsFactTable* and a new token is pushed onto the stack.

Section 4.2.4 briefly mentioned the "sweep" stage involved in the reasoning process. It has been stated above how this is dealt with at Graph level. At LEAPS level, this

involves iterating through all instances of *LeapsFactTable* and removing any fact-handle that has a time-stamp that is below the current time. The corresponding token must also be removed from the stack.

5.3.5 Difficulties Encountered

The removal of fact-handles during the sweep stage was initially problematic. While iterating over the *LeapsFactTable*, a removal would cause the iterator to throw an exception as it tried to move onto a recently removed element. A simple solution for this problem was to add each element into a temporary collection and then remove each element individually once all elements that were to be removed were known. Removing an element from a collection once iterating over the collection has finished allows for the safe removal of the element.

To determine if a triple already exists in a *LeapsFactTable*, a simple hashing technique was used as mentioned earlier. Using the new triple object a unique hash for the subject, predicate and object of the triple is obtained. This can then be used as a key in a hashtable in order to determine if an existing matching triple exists in the reasoner and to retrieve the existing triple if necessary.

The implementation of functions as condition elements also proved difficult. Various rules used in SCOROR make use of functions as condition elements. A simple implementation was attempted and worked for very small ontologies. However, for larger ontologies the implementation dramatically reduced reasoning times. The reason for this is still unknown, but further work could expand on this implementation.

5.4 Summary

This chapter has discussed the implementation of the LEAPS approach. The concepts which were introduced in the Design chapter were described in further detail and further examples were given.

The next chapter will assess this implementation of a LEAPS reasoner in comparison to the existing RETE reasoner in SCOROR.

Chapter 6

Evaluation

6.1 Introduction

Evaluating a stream reasoner is a complicated task due to the high amount of variables associated with the process. Hardy [14] performed a series of experiments in order to gain an understanding of how different factors influenced SCOROR's performance. In the interest of comparing the implementation of the RETE algorithm to the chosen algorithm accurately, it must be assumed that the same rule set and ontologies used in the evaluation are identical. Unfortunately, some rules which include functions could not be included in experiments as the LEAPS implementation does not fully support functions yet. Another factor to consider is the varying nature of performance on different machines. Results can vary significantly. As a result of the above facts this evaluation has performed the following experiments on both the RETE and LEAPS implementation in SCOROR in order to give a more authentic comparison.

Initial experiments to ensure the correct operation of the LEAPS implementation were also performed. These experiments are not documented here, but were conducted in order to ensure that both reasoners generated identical deductions given identical rule-sets and facts.

All experiments were performed on a Dell XPS L502X. Memory: 6144MB RAM, Processor: Intel Core i5-2450M CPU, 2.5GHz.

The rule set used is given in Appendix A.

6.2 Experiments

Initial experiments performed by Hardy made an attempt to measure the throughput of SCOROR. Throughput is an important factor as the reasoner must be able to complete reasoning over the recently inserted triples before the next window of triples is available to be inserted. If the reasoner fails to complete the reasoning in this period, the number of triples which get inserted on the next iteration will be increased (assuming a constant stream speed). When these triples are inserted, the reasoning time will only increase resulting in a continuous cycle of reduced performance. This is referred to as the reasoner becoming overloaded.

These initial experiments clearly illustrated that the number of triples needed to overload the reasoner varied depending on the ontology used.

Further experiments illustrated that the re-reasoning time (reasoning time for an iteration) varied based on a number of factors. The most notable factors included initial static ontology size and window size. The reasons behind these factors are quite logical. A larger ontology size results in a more facts permanently contained in the reasoner. A larger window size results in a slow collection of temporal facts which are eventually removed, only to be replaced by more temporal facts.

The conclusion of the experiments by Hardy is that each of these factors must be considered with relevance to the setting in which the reasoner is to be employed. The amount of predefined data (static ontology size), reducing relevance of data (window size), input of data (peak stream speed) and reasoning outcome trends should be examined in order to determine whether the reasoner can adequately reason over the incoming data and the reasoner's overall capability for a particular setting.

The experiments have been performed again not in an attempt to determine conclusions which challenge those put forward by Hardy but in order to compare the two implementations, RETE and LEAPS. All times are given in milliseconds.

6.2.1 Experimental Ontologies

Various ontologies were used in experiments in order to truly compare the reasoning capabilities of both implementations. This is due to the fact that ontologies can vary in both size and expressivity, which can be significant factors in reasoning times.

The ontologies used in the experiments are listed in Table 6.1. These ontologies were used in the experiments by Hardy and were taken from Tai et al [5].

Ontology	Expressiveness	Size (Triples)
Teams	ALCIN	87
Beer	ALHI(D)	173
Mindswapper	ALCHIF(D)	437
Mad_cows	ALCHOIN(D)	521
Biopax	ALCHF(D)	633
Food	ALCOF	924
University	SIOF(D)	169
Koala	ALCON(D)	147

TABLE 6.1: Experimental Ontologies

A full explanation of the expressivity naming convention can be found at [30].

6.2.2 Throughput Experiment

The throughput of the reasoners was examined using a window size of 1 second. In this case, it is required that re-reasoning is completed within 1 second before the next window is available for insertion. The stream speed was increased gradually so that the number of triples inserted rose in a sequence of steps with the intention that the re-reasoning time would pass the 1 second mark. This experiment was performed with two ontologies, the Teams and Biopax ontologies. These ontologies were used as they represent a rather simple ontology (Teams) and a more complex ontology (Biopax).

Figure 6.1 and Figure 6.2 show the throughput for the Teams ontology for both the RETE and LEAPS implementations respectively. Figure 6.3 and Figure 6.4 show the throughput for the Biopax ontology. In the following charts re-reasoning time for a

particular iteration is given on the y-axis in milliseconds. This is the time taken to complete reasoning over the recently inserted triples. The number of triples inserted for this iteration is given on the x-axis.

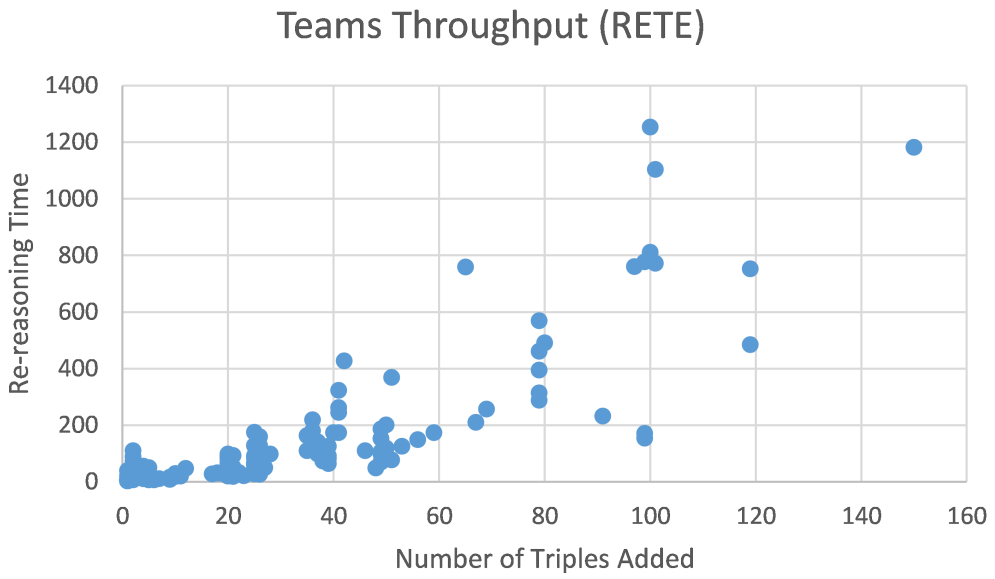

FIGURE 6.1: RETE Teams Throughput

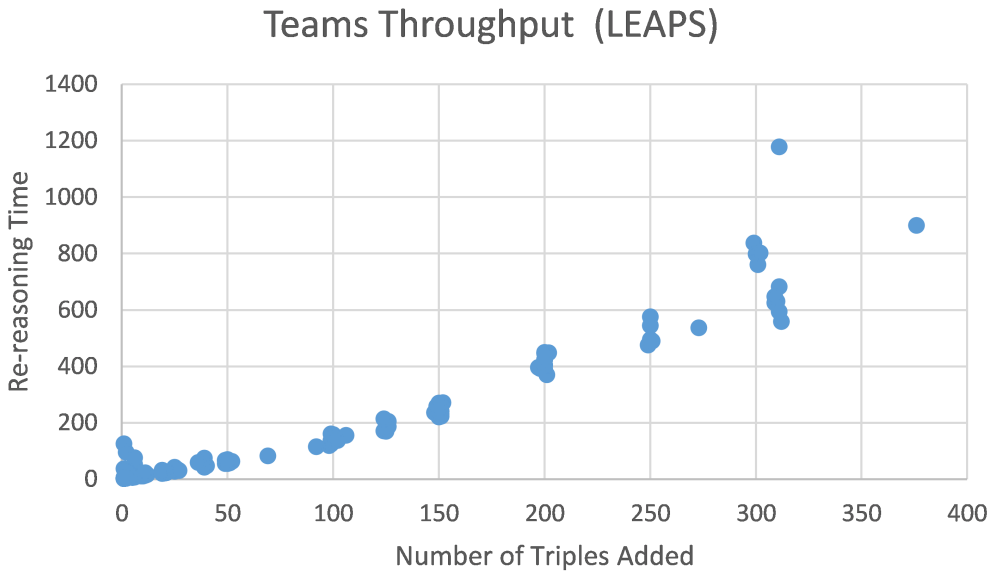

FIGURE 6.2: LEAPS Teams Throughput

The re-reasoning time can be seen to vary for the same number of triples inserted. It should be made clear that the work that needs to be performed will vary based on the actual triples that are inserted and is not solely based on the number of triples inserted.

The data used in the figures has been collected over a number of iterations in order to give a good sample of re-reasoning times.

Figure 6.1 shows the 1 second re-reasoning time is breached at around 100 insertions for the RETE implementation. Figure 6.2 shows the 1 second re-reasoning time is breached at around 300 insertions for the LEAPS implementation.

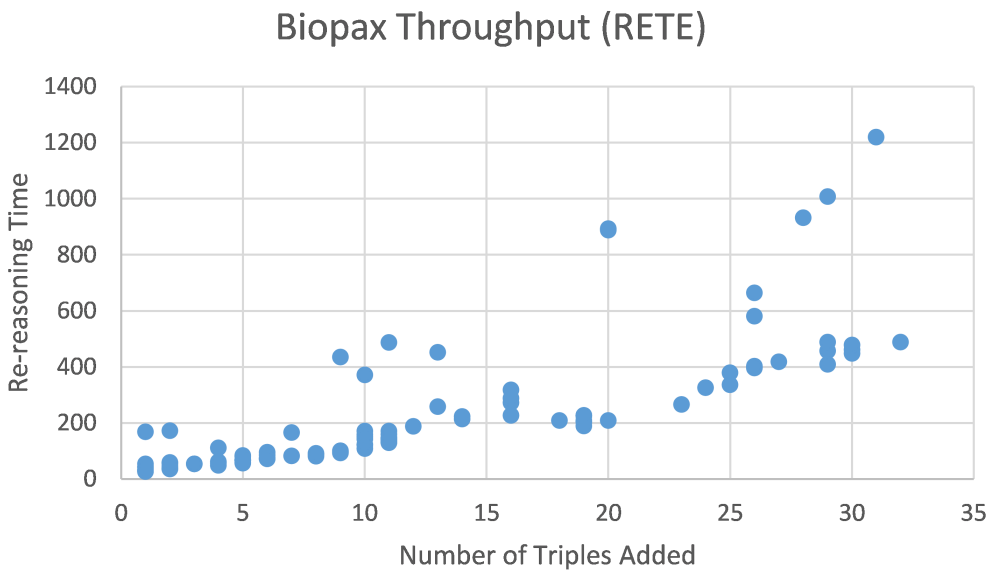

FIGURE 6.3: RETE Biopax Throughput

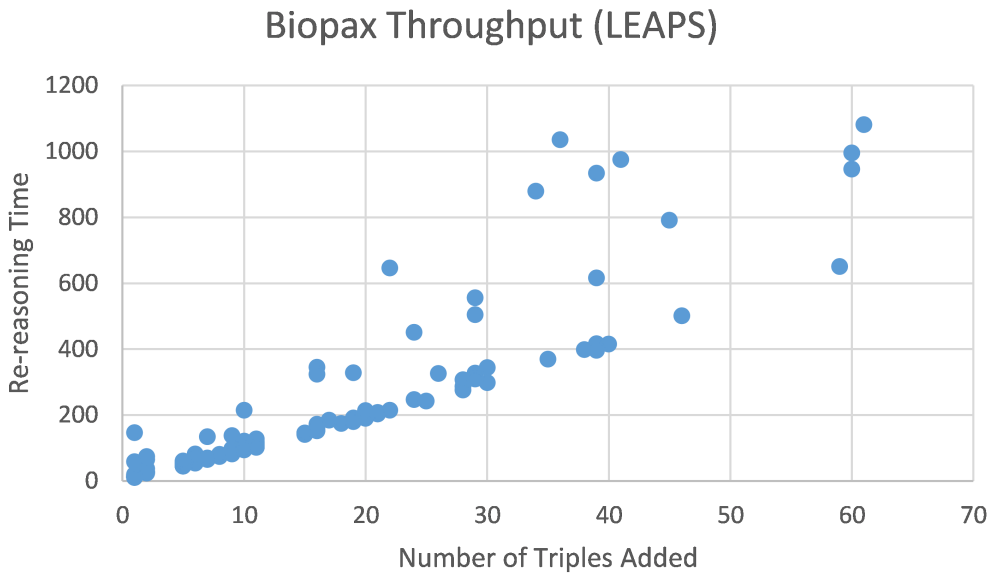

FIGURE 6.4: LEAPS Biopax Throughput

Figure 6.3 shows the 1 second re-reasoning time is breached at around 28 insertions for the RETE implementation. Figure 6.4 shows the 1 second re-reasoning time is breached at around 36 insertions for the LEAPS implementation.

From the above figures it can be clearly seen that both reasoners offer a linear relationship between re-reasoning time and the number of triples that are inserted. The LEAPS implementation performs better than the RETE implementation in the two cases given. The RETE reasoner becomes overloaded with a smaller number of insertions than the LEAPS reasoner in both cases.

6.2.3 Window Variability Experiment

Window size directly influences the number of the temporal triples that are stored in the reasoner at any given time. A larger window size will result in temporal triples being contained for a longer period. This obviously results in a larger working memory and results in a longer re-reasoning time.

This experiment was conducted by setting a fixed stream speed of 10 triples per second. The ontology used was the Teams ontology. The window size was then tested at 2, 5, 10 and 20 seconds.

Figure 6.5 and Figure 6.6 show the results for the Teams ontology for both the RETE and LEAPS implementations respectively. In the following charts re-reasoning time for a particular iteration is given on the y-axis in milliseconds. This is the time taken to complete reasoning on the recently inserted triples. The number of triples inserted for this iteration is given on the x-axis.

Both figures show quite disperse reasoning times for the same number of triples added. Again, the amount of reasoning that is performed will vary based on the actual triples that are inserted and is not solely based on the number of triples inserted.

It is difficult to identify any clear trends in reasoning times with respect to window size in the RETE implementation. However, while there are clear outliers in the LEAPS implementation, there is also a clear trend. The 2 and 5 second windows seem to offer similar reasoning times while the 10 second window offers slightly longer reasoning times. The rise in reasoning times for the 20 second window is more noticeable.

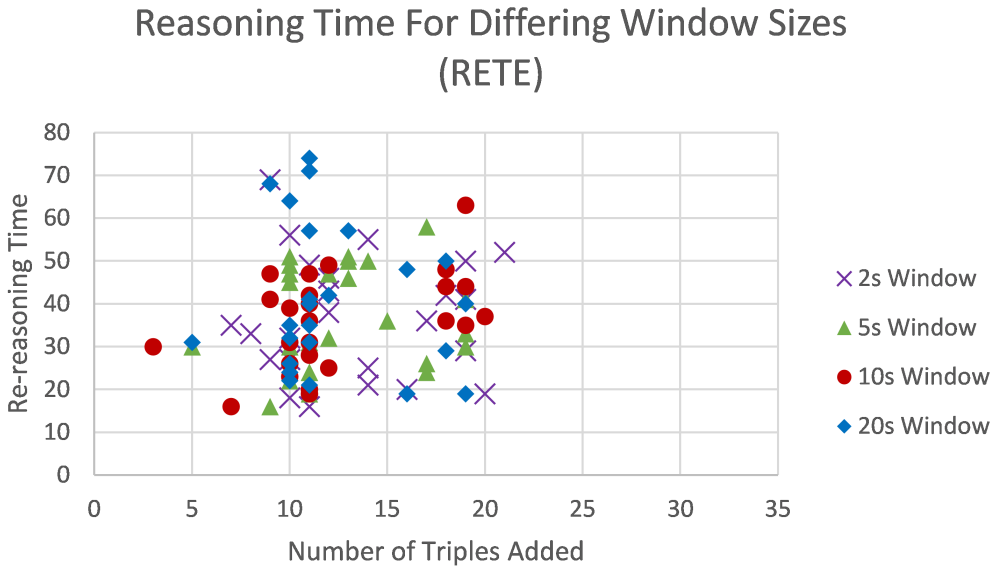

FIGURE 6.5: RETE Window Sizes

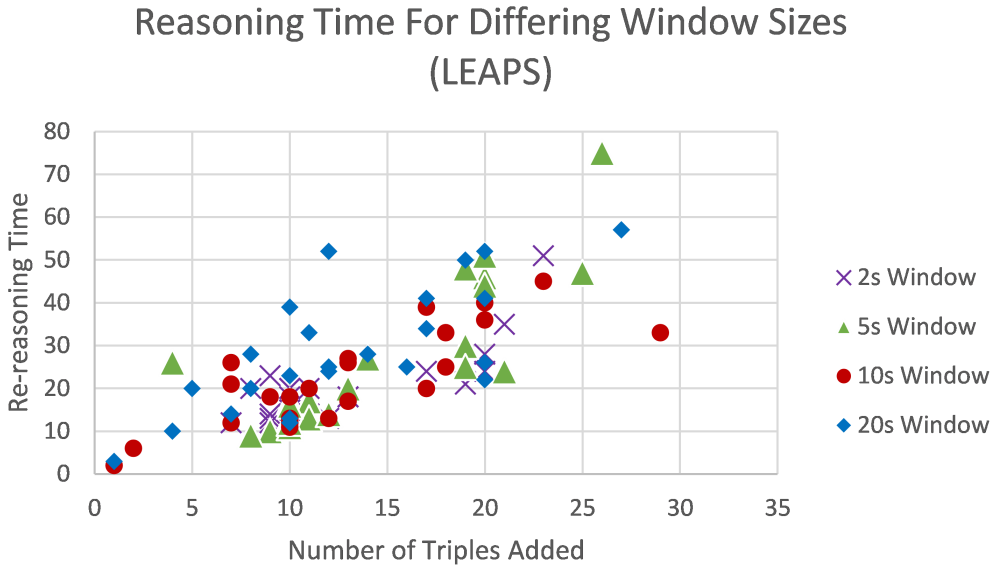

FIGURE 6.6: LEAPS Window Sizes

In general, the LEAPS implementation again shows lower reasoning times.

6.2.4 Differing Ontology Experiment

Hardy found that static ontology size had a clear impact on re-reasoning times. In order to compare the two implementations, the average re-reasoning time for all ontologies was

obtained over 13 iterations. This experiment was conducted by setting a fixed stream speed of 10 triples per second and a fixed window size of 10.

Figure 6.7 illustrates the results. Average re-reasoning time over 13 iterations is given on the y-axis in milliseconds.

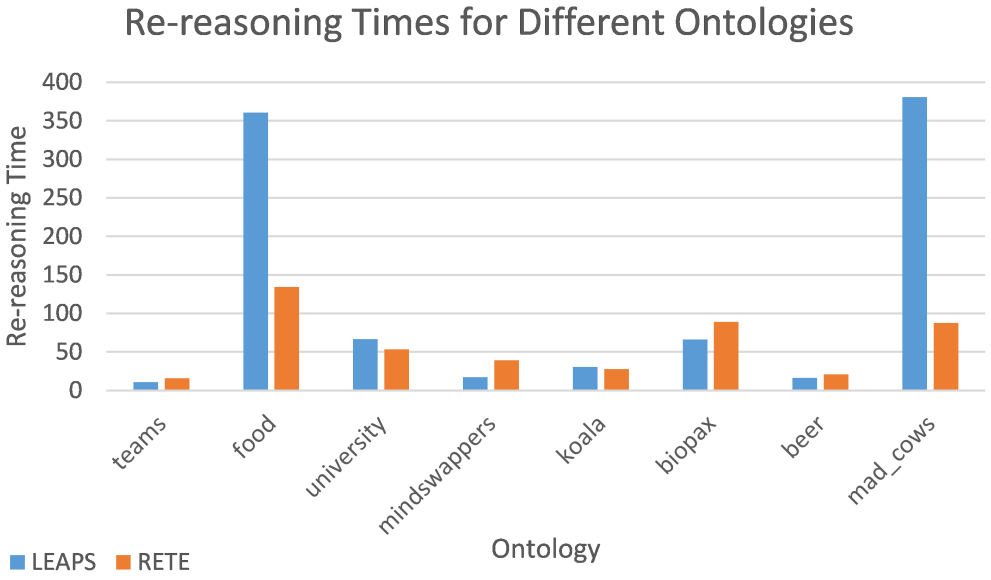

FIGURE 6.7: RETE vs LEAPS Ontology Re-reasoning Times

Figure 6.8 illustrates the results with respect to ontology size.

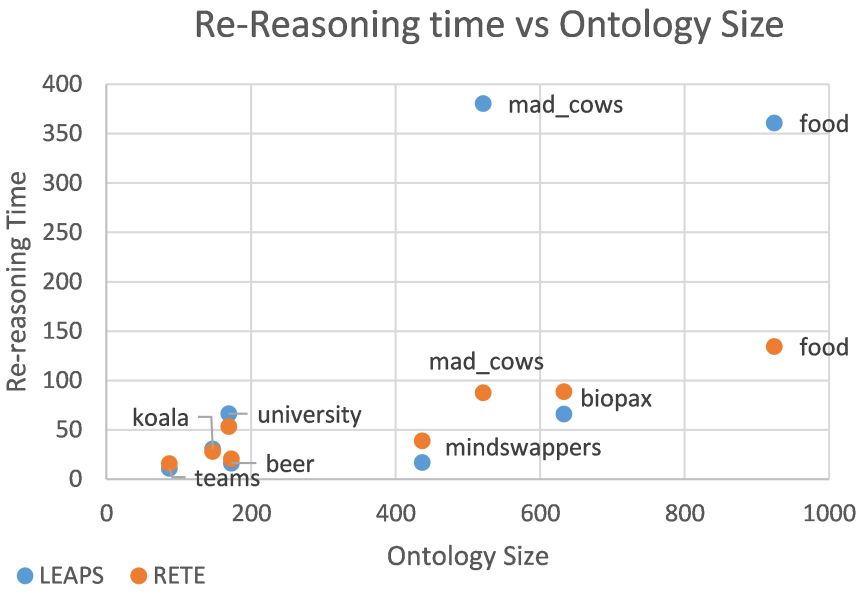

FIGURE 6.8: Re-Reasoning Time vs Ontology Size

It is quite clear that the RETE implementation outperforms LEAPS by a large margin for both the food and mad_cows ontologies. The reason for this seems to be the combination of a large ontology size and a high expressivity. While Hardy determined that expressivity was not a large factor in the reasoning time, it is clear that for the LEAPS implementation it seems to play a large part when coupled with a large ontology size.

The other ontologies all have similar reasoning times with LEAPS performing better on the teams and beer ontologies and more noticeably on the mindswappers and biopax ontologies. RETE performs slightly better on both the koala and university ontologies.

The explanation put forward is that the LEAPS approach is not designed for use with wild-card predicates. When a condition element contains a wild-card predicate all facts in the working memory must be considered. Clearly this significantly effects reasoning time as all facts must be iterated over for each condition element that contains a wild-card predicate. High expressivity can result in changes propagating widely through an ontology due to properties such as transitivity. The coupling of these two factors has a clear effect on the reasoning times in the LEAPS implementation.

A second factor that may have a consequence on the reasoning time is the resetting of the iteration state for a rule-handle. The Implementation chapter describes how the iteration state may be reset if it is detected that the state points to a fact that has been removed since the last search. Since temporal triples are removed on every cycle after the cycle which is numbered the window size, this could be a factor in the reasoning time for larger ontologies.

6.2.5 Memory Consumption Experiment

Clearly the memory consumption of the reasoner is quite important in resource constrained devices. The main inspiration for this research was to determine whether a suitable replacement for the RETE algorithm could be found. Thus an experiment to measure the memory consumption of the reasoner was performed in order to assess whether the LEAPS algorithm offers better performance in this area.

This experiment was conducted by setting a fixed stream speed of 10 triples per second and a fixed window size of 10. All ontologies were tested.

Figure 6.9 illustrates the results. Average memory consumption over 25 iterations is given on the y-axis in bytes.

Figure 6.10 illustrates the results with respect to ontology size.

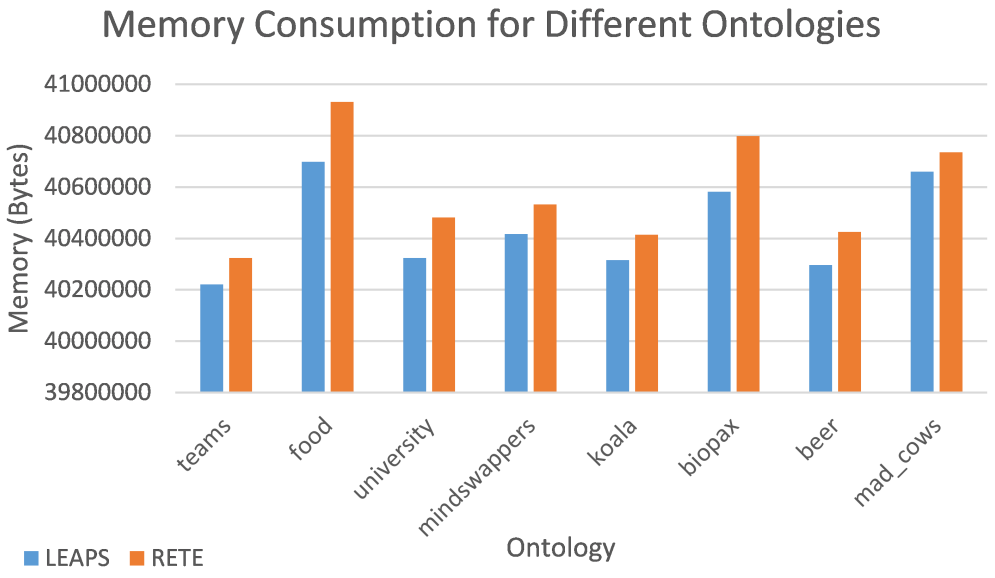

FIGURE 6.9: RETE vs LEAPS Ontology Memory Consumption

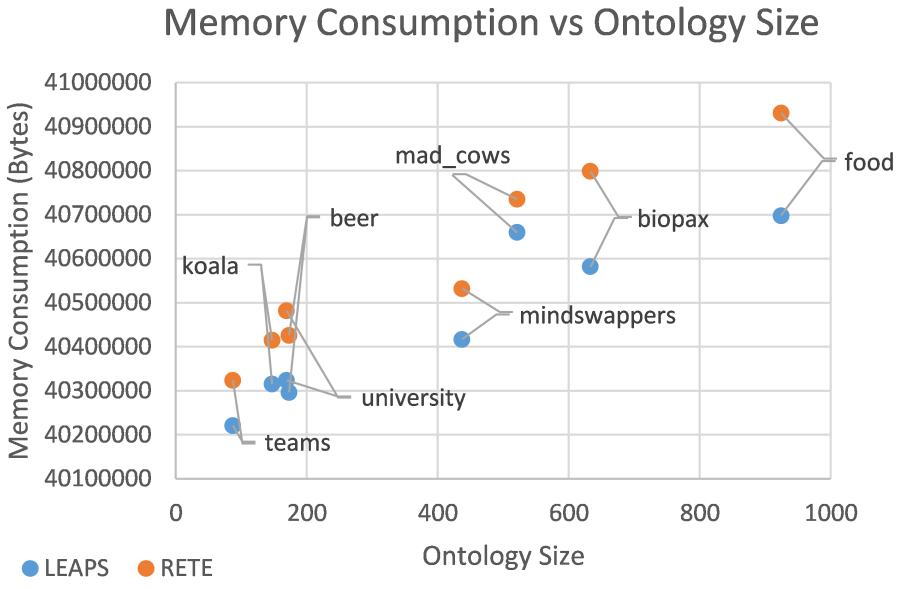

FIGURE 6.10: Memory Consumption vs Ontology Size

The results reinforce the conclusions of Hardy that original ontology size has a large part to play in memory consumption. There is an obvious linear relationship between

ontology size and memory consumption. The results also clearly show that LEAPS outperforms RETE for all ontologies. It should be noted that the majority of memory consumption in SCOROR is due to the use of the C-SPARQL stream processor. In the interest of comparing the two reasoners more clearly, the average memory consumption of the stream processor was also recorded and deducted from the total value. Figure 6.11 illustrates these results. Figure 6.12 illustrates the LEAPS memory consumption for each ontology as a percentage of the RETE memory consumption.

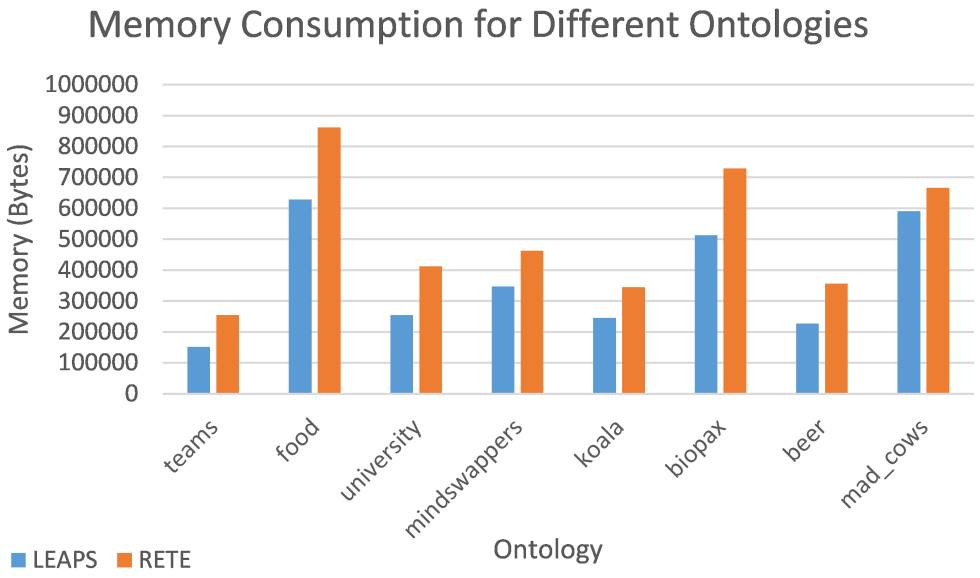

FIGURE 6.11: RETE vs LEAPS Ontology Memory Consumption (Isolated Reasoner)

These results highlight the difference in memory consumptions somewhat more clearly. There is a significant difference in most cases. Such reductions could be vital in a resource constrained device.

6.3 Summary

A direct comparison between the existing RETE stream reasoner and the newly implemented LEAPS stream reasoner has been performed in this chapter. Experiments considered the number of triples inserted into the reasoner at each reasoning cycle, the effect of the window size, and the base ontology used for reasoning.

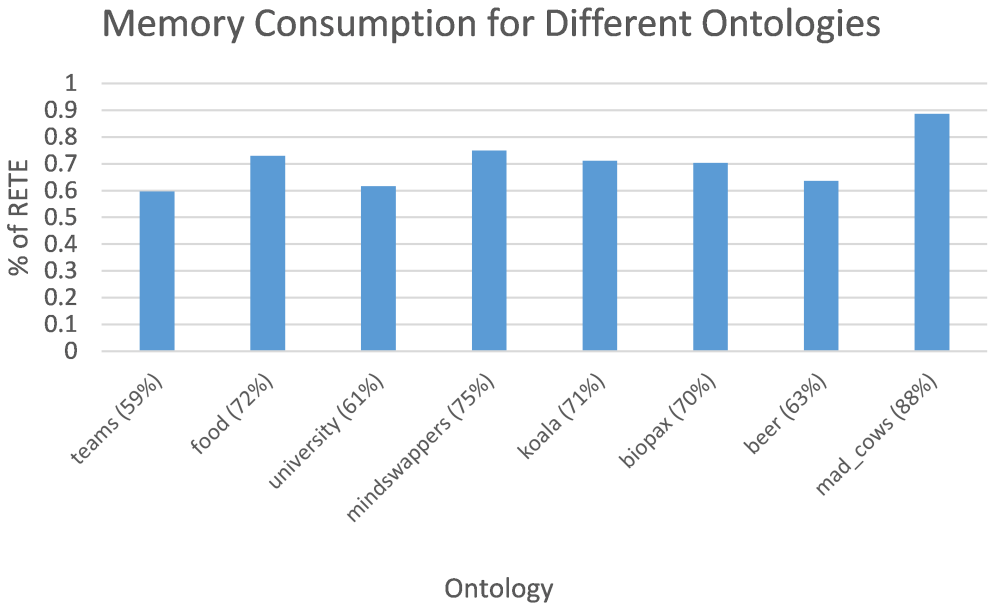

FIGURE 6.12: LEAPS Memory Consumption as percentage of RETE Memory Consumption

Both reasoners offer a linear relationship between re-reasoning time and the number of triples that are inserted. The LEAPS implementation performs better than the RETE implementation for the two ontologies tested.

The LEAPS implementation offers a clear trend in reasoning times based on the varying window size. This trend is not so clear in the RETE implementation. This suggests that the LEAPS implementation may offer a more reliable performance.

It is quite clear that the RETE implementation outperforms LEAPS by a large margin when there is a combination of a large ontology size and a high expressivity. Reasons for this have been explained in detail above and can be summarised as the use of wild-card predicates and the resetting of saved iteration states due to the removal of temporal triples. Given further time, future experiments would be performed to examine the effect of these factors in the throughput and window variability experiments.

The LEAPS implementation offers a clear improvement in terms of memory consumption. The best case measured offered a 40% improvement, while the worst case offered 11%, still a significant improvement. Such reductions could be vital in a resource constrained device.

Table 6.2 offers a brief summary of the results in an easily digestible manner.

Experiment	Variable	Outcome
Throughput	Number of triples inserted	RETE reasoner becomes overloaded with a smaller number of insertions than LEAPS reasoner.
Window Variability	Window Size	Clear trends in reasoning times with respect to window size observed in LEAPS, unclear with RETE.
Differing Ontology	Base Ontology	Similar reasoning times for ontologies (exception of large size and high expressivity).
Memory Consumption	Base Ontology	LEAPS reasoner offers significant memory reductions over RETE reasoner.

TABLE 6.2: Summary of Experiments

Chapter 7

Conclusion

7.1 Objectives Achieved

The first goal of this research was to determine which of two possible algorithms would be more suitable for use in a stream reasoning system with memory constraints. The primary goal was then to determine whether an implementation of the chosen algorithm provided a more efficient option in terms of memory consumption in a resource constrained device in comparison to an existing implementation of the RETE algorithm.

The State of the Art research performed highlighted the two algorithms as TREAT, a memory conscious adaption of RETE, and LEAPS, a lazy evaluation approach for reasoning. It was noted that LEAPS would be significantly more difficult to implement, but it was chosen as the algorithm to be implemented. This was due to the fact that evidence suggested that the approach could provide stronger performance guarantees over RETE and TREAT.

The paper describes the design and implementation of the LEAPS approach based on the DROOLS [29] design, and the modification of this implementation in order to support stream reasoning. As of writing, this is the first known adaption of the LEAPS algorithm into a stream reasoning environment.

The implementation was evaluated in SCOROR [14] as a possible replacement for a RETE based reasoner developed by Tai et al [5].

Evaluation of the LEAPS reasoner found that the reasoning times for most base ontologies are quite similar to that of the RETE reasoner. Notable exceptions are when there is a combination of a large ontology size with high expressivity. The suggested reasons for this have been explained in the Evaluation chapter as the use of wild-card predicates and the resetting of saved iteration states due to the removal of temporal triples.

Memory consumption for the LEAPS reasoner was shown to be significantly reduced with comparison to the RETE reasoner. As predicted, memory consumption increases with base ontology size.

7.2 Contribution of Research

Analysis and comparison of both TREAT and LEAPS has been offered and the outcome illustrates that both algorithms would be favourable above RETE for stream reasoning on memory constrained devices.

This research has also shown that the LEAPS reasoning approach can be modified to perform in a streaming environment. It also has demonstrated how it can be implemented to provide a memory efficient option for stream reasoning on resource constrained devices.

The evaluation of this implementation of the LEAPS approach has highlighted how the LEAPS reasoning approach may not be ideal in situations when wild-card predicates are being used in rules. Large ontologies combined with wild-card predicates will result in all facts in working memory being considered for any condition element with a wild-card predicate.

7.3 Future Work

7.3.1 Negative Condition Support

Rules used in SCOROR and the research performed by Hardy [14] did not support the use of negative condition elements. For this reason, and in the interest of comparing the implementation of the RETE algorithm to the LEAPS implementation accurately, negative condition support was not included in this research implementation of LEAPS.

However, the requirements for negative condition support have been discussed in the State of the Art chapter. Further work could see the implementation of these requirements so that additional rules could be included in the reasoning process. This would allow for the use of alternative ontologies which allow for further use of OWL specifications.

7.3.2 Function Support

Similar to the above functionality, function support could be included in condition elements. A naive approach to the inclusion of functions was implemented for this research but proved too slow for larger ontologies. Further work could expand on this implementation so that functions may be included.

7.3.3 Indexing

For the removal of temporal triples SCOROR must search through all facts in the graph and also in the *LeapsFactTable* instances. The time consumed for this search could be reduced using some sort of indexing and could result in a reduction in reasoning time.

A simple example could be to place temporal triples in a collection based on the time-stamp and simply remove this collection when the time-stamp expires.

7.4 Final Remarks

The research question put forward by this paper is again given below.

> "In comparison to an existing implementation of the RETE algorithm, can a more efficient approach, in terms of memory consumption, be developed for stream reasoning on resource constrained devices?"

This research investigated two possible alternatives to the RETE algorithm. The TREAT and LEAPS approaches were discussed and compared in detail. LEAPS was chosen as

the approach to be implemented due to the the conclusion that it could provide stronger performance guarantees over RETE and TREAT.

The LEAPS implementation included modifications in order to support stream reasoning efficiently. Evaluation of the LEAPS reasoner found that the reasoning times for most base ontologies were quite similar to that of the RETE reasoner, with some exceptions. Memory consumption of the LEAPS reasoner always outperformed the RETE reasoner by some margin.

Thus several factors should be considered when determining what reasoner to use in SCOROR. If re-reasoning time is the most important factor and the base ontology is large and complex, perhaps RETE is the best option. However, the primary goal of this research was to determine whether a more efficient option in terms of memory consumption in a resource constrained device could be found.

This research has shown that the LEAPS algorithm can be implemented and modified to perform in a streaming environment. Further evaluation suggests that it is indeed the case that a more efficient option in terms of memory consumption exists in the form of this implementation.

Appendix A

Rule Set

[rdf1: (?v ?p ?w) ->(?p rdf:type rdf:Property)]

[rdfs2: (?p rdfs:domain ?u), (?v ?p ?w) ->(?v rdf:type ?u)]

[rdfs4a: (?v ?p ?w) ->(?v rdf:type rdfs:Resource)]

[rdfs5: (?v rdfs:subPropertyOf ?w), (?w rdfs:subPropertyOf ?u) ->(?v rdfs:subPropertyOf ?u)]

[rdfs6: (?v rdf:type rdf:Property) ->(?v rdfs:subPropertyOf ?v)]

[rdfs7x: (?p rdfs:subPropertyOf ?q), (?v ?p ?w) ->(?v ?q ?w)]

[rdfs8: (?v rdf:type owl:Class) ->(?v rdfs:subClassOf rdfs:Resource)]

[rdfs9: (?v rdfs:subClassOf ?w), (?u rdf:type ?v) ->(?u rdf:type ?w)]

[rdfs10: (?v rdf:type owl:Class) ->(?v rdfs:subClassOf ?v)]

[rdfs11: (?v rdfs:subClassOf ?w), (?w rdfs:subClassOf ?u) ->(?v rdfs:subClassOf ?u)]

[rdfs12: (?v rdf:type rdfs:ContainerMembershipProperty) ->(?v rdfs:subPropertyOf rdfs:member)]

[rdfs13: (?v rdf:type rdfs:Datatype) ->(?v rdfs:subClassOf rdfs:Literal)]

[rdfp2: (?p rdf:type owl:InverseFunctionalProperty), (?u ?p ?w), (?v ?p ?w) ->(?u owl:sameAs ?v)]

[rdfp4: (?p rdf:type owl:TransitiveProperty), (?u ?p ?v), (?v ?p ?w) ->(?u ?p ?w)]

[rdfp5a: (?v ?p ?w) ->(?v owl:sameAs ?v)]

[rdfp7: (?u owl:sameAs ?v), (?v owl:sameAs ?w) ->(?u owl:sameAs ?w)]

[rdfp9: (?v rdf:type owl:Class), (?v owl:sameAs ?w) ->(?v rdfs:subClassOf ?w)]

[rdfp10: (?p rdf:type rdf:Property), (?p owl:sameAs ?q) ->(?p rdfs:subPropertyOf ?q)]

[rdfp12a: (?v owl:equivalentClass ?w) ->(?v rdfs:subClassOf ?w)]

[rdfp12c: (?v rdfs:subClassOf ?w), (?w rdfs:subClassOf ?v) ->(?v owl:equivalentClass ?w)]

[rdfp13a: (?v owl:equivalentProperty ?w) ->(?v rdfs:subPropertyOf ?w)]

[rdfp13c: (?v rdfs:subPropertyOf ?w), (?w rdfs:subPropertyOf ?v) ->(?v owl:equivalentProperty ?w)]

[rdfp14a: (?v owl:hasValue ?w), (?v owl:onProperty ?p), (?u ?p ?w) ->(?u rdf:type ?v)]

[rdfp15: (?v owl:someValuesFrom ?w), (?v owl:onProperty ?p), (?x rdf:type ?w), (?u ?p ?x) ->(?u rdf:type ?v)]

Bibliography

[1] Ping Wang, Jin Guang Zheng, Linyun Fu, Evan W. Patton, Timothy Lebo, Li Ding, Qing Liu, Joanne S. Luciano, and Deborah L. McGuinness. A Semantic Portal for Next Generation Monitoring Systems. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part II*, ISWC'11, pages 253–268, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25092-7. URL `http://dl.acm.org/citation.cfm?id=2063076.2063094`.

[2] P. Anatharam P. Desai, C. Henson and A. Sheth. Demonstration: SECURE – Semantics Empowered resCUe Environment. In *Proceedings of the 4th International Workshop on Semantic Sensor Network*, pages 115–118, 2011. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.366.2559`.

[3] Tommaso Di Noia, Roberto Mirizzi, Vito Claudio Ostuni, Davide Romito, and Markus Zanker. Linked Open Data to Support Content-based Recommender Systems. In *Proceedings of the 8th International Conference on Semantic Systems*, I-SEMANTICS '12, pages 1–8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1112-0. doi: 10.1145/2362499.2362501. URL `http://doi.acm.org/10.1145/2362499.2362501`.

[4] Marta Sabou, Adrian M. P. Braşoveanu, and Irem Arsal. Supporting Tourism Decision Making with Linked Data. In *Proceedings of the 8th International Conference on Semantic Systems*, I-SEMANTICS '12, pages 201–204, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1112-0. doi: 10.1145/2362499.2362533. URL `http://doi.acm.org/10.1145/2362499.2362533`.

[5] W. Tai, D. O'Sullivan, and J. Keeney. Resource Constrained Reasoning Using a Reasoner Composition Approach. 2013. URL http://www.semantic-web-journal.net/system/files/swj545.pdf.

[6] Ossama Younis, Sonia Fahmy, and Paolo Santi. An Architecture for Robust Sensor Network Communications. 2005. URL https://www.cs.purdue.edu/homes/fahmy/papers/reed-journal.pdf.

[7] Safdar Ali and Stephan Kiefer. μOR — A Micro OWL DL Reasoner for Ambient Intelligent Devices. In *Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing*, GPC '09, pages 305–316, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-01670-7. doi: 10.1007/978-3-642-01671-4_28. URL http://dx.doi.org/10.1007/978-3-642-01671-4_28.

[8] Christian Seitz and René Schönfelder. Rule-based OWL Reasoning for Specific Embedded Devices. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part II*, ISWC'11, pages 237–252, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25092-7. URL http://dl.acm.org/citation.cfm?id=2063076.2063093.

[9] Daniel P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. In *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*, AAAI'87, pages 42–47. AAAI Press, 1987. ISBN 0-934613-42-7. URL http://dl.acm.org/citation.cfm?id=1856670.1856678.

[10] Don Batory. The LEAPS Algorithms. Technical report, Austin, TX, USA, 1994. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.8595.

[11] Davide Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Stream Reasoning: Where We Got So Far. In *Proceedings of the NeFoRS2010 Workshop, co-located with ESWC2010*, 2010. URL http://wasp.cs.vu.nl/larkc/nefors10/paper/nefors10_paper_0.pdf.

[12] Fredrik Heintz. Semantically Grounded Stream Reasoning Integrated with ROS. *Intelligent Robots and Systems (IROS)*, pages 5935–5942, 2013. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.395.6314.

[13] Spyros Kotoulas Freddy Lecue and Pol Mac Aonghusa. Capturing the Pulse of Cities: A Robust Stream Data Reasoning Approach. *IBM Research Smarter Cities Technology Centre*, 2011. URL `http://wiki.planet-data.eu/uploads/d/d3/Capturing_the_Pulse_of_Cities,_Freddy_Lecue.pdf`.

[14] C. E. Hardy. Stream Reasoning on Resource-Limited Devices. *Trinity College Dublin*, 2013. URL `https://www.scss.tcd.ie/~osulldps/colinhardy.pdf`.

[15] Charles L. Forgy. Expert Systems. chapter Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, pages 324–341. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990. ISBN 0-8186-8904-8. URL `http://dl.acm.org/citation.cfm?id=115710.115736`.

[16] W3C OWL Overview. `http://www.w3.org/2001/sw/wiki/OWL`. [Online; accessed 21-May-2014].

[17] W3C RDF Overview. `http://www.w3.org/RDF`, . [Online; accessed 21-May-2014].

[18] RDF Vocabulary Description Language 1.0: RDF Schema. `http://www.w3.org/TR/rdf-schema`, . [Online; accessed 21-May-2014].

[19] J. McDermott, A. Newell, and J. Moore. The Efficiency of Certain Production System Implementations. *SIGART Bull.*, (63):38–38, June 1977. ISSN 0163-5719. doi: 10.1145/1045343.1045366. URL `http://doi.acm.org/10.1145/1045343.1045366`.

[20] SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`. [Online; accessed 21-May-2014].

[21] C-SPARQL: SPARQL for Continuous Querying. `http://wiki.larkc.eu/c-sparql`. [Online; accessed 21-May-2014].

[22] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying rdf streams with c-sparql. *SIGMOD Rec.*, 39(1): 20–26, September 2010. ISSN 0163-5808. doi: 10.1145/1860702.1860705. URL `http://doi.acm.org/10.1145/1860702.1860705`.

[23] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Incremental Reasoning on Streams and Rich Background Knowledge. In *Proceedings of the 7th International Conference on The Semantic Web: Research and Applications - Volume Part I*, ESWC'10, pages 1–15, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13485-8, 978-3-642-13485-2. doi: 10.1007/978-3-642-13486-9_1. URL `http://dx.doi.org/10.1007/978-3-642-13486-9_1`.

[24] Ian Wright and James A. R. Marshall. The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case. *Int. J. Intell. Games & Simulation*, 2 (1):36–48, 2003. URL `http://dblp.uni-trier.de/db/journals/ijigs/ijigs2.html#WrightM03`.

[25] Daniel P. Miranker, David A. Brant, Bernie Lofaso, and David Gadbois. On the Performance of Lazy Matching in Production Systems. In *Proceedings of the Eighth National Conference on Artificial Intelligence - Volume 1*, AAAI'90, pages 685–692. AAAI Press, 1990. ISBN 0-262-51057-X. URL `http://dl.acm.org/citation.cfm?id=1865499.1865602`.

[26] Don Batory, Jeff Thomas, and Marty Sirkin. Reengineering a Complex Application Using a Scalable Data Structure Compiler. In *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '94, pages 111–120, New York, NY, USA, 1994. ACM. ISBN 0-89791-691-3. doi: 10.1145/193173.195299. URL `http://doi.acm.org/10.1145/193173.195299`.

[27] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental Pattern Matching in the Viatra Model Transformation System. In *Proceedings of the Third International Workshop on Graph and Model Transformations*, GRaMoT '08, pages 25–32, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-033-3. doi: 10.1145/1402947.1402953. URL `http://doi.acm.org/10.1145/1402947.1402953`.

[28] Daniel P. Miranker and Lance Obermeyer. An Overview of the VenusDB Active Multidatabase System. In *In Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications*, pages 664–682. Addison-Wesley/ACM Press, 1996.

[29] DROOLS: Business Rule Management System. `https://www.jboss.org/drools/`. [Online; accessed 21-May-2014].

[30] Sebastian Rudolph: Foundations of Description Logics. `http://www.aifb.kit.edu/images/1/19/DL-Intro.pdf`. [Online; accessed 21-May-2014].