A DISTRIBUTED RULE-BASED EXPERT SYSTEM FOR LARGE EVENT STREAM PROCESSING

by

YI CHEN

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
January 2019

Abstract

Rule-based expert systems (RBSs) provide an efficient solution to many problems that involve event stream processing. With today's needs to process larger streams, many approaches have been proposed to distribute the rule engines behind RBSs. However, there are some issues which limit the potential of distributed RBSs in the current big data era, such as the load imbalance due to their distribution methods, and low parallelism originated from the continuous operator model.

To address these issues, we propose a new architecture for distributing rule engines. This architecture adopts the *dynamic job assignment* and the *micro-batching* strategies, which have recently arisen in the big data community, to remove the load imbalance and increase parallelism of distributed rule engines. An automated transformation framework based on Model-driven Architecture (MDA) is presented, which can be used to transform the current rule engines to work on the proposed architecture. This work is validated by a 2-step verification.

In addition, we propose a generic benchmark for evaluating the performance of distributed rule engines. The performance of the proposed architecture is discussed and directions for future research are suggested.

The contribution of this study can be viewed from two different angles: for the rule-based system community, this thesis documents an improvement to the rule engines by fully adopting big data technologies; for the big data community, it is an early proposal to process large event streams using a well crafted rule-based system. Our results show the proposed approach can benefit both research communities.

ACKNOWLEDGEMENTS

I would like to thank my supervisors Prof. Peter Tiňo and Dr. Behzad Bordbar for their support and constructive suggestions during my PhD research. Financial support from the School of Computer Science in the form of a faculty scholarship is deeply appreciated.

Special thanks to Mr. Keith Harrison from HP Labs, for his help in the early stage of my research.

I'd like to thank my friends and colleagues in the School of Computer Science, University of Birmingham, for their support during these years.

Finally, I wish to express my love to my family for the support and encouragement.

CONTENTS

LIST OF FIGURES

LISTINGS

CHAPTER 1

INTRODUCTION

Rule-based systems (RBSs) have been studied comprehensively since the 1980s in the realms of expert systems. They have been used widely in diverse domains such as electronic, communication and enterprise systems [16, 23, 79]. The main goal of rule-based systems is the separation of business logics from system implementations, which enables domain experts to construct and maintain software systems without the knowledge of programming [34, 56]. In addition, with computer scientists concentrating on building efficient rule engines, these rule-based systems can achieve reasonably high performance.

A typical application of rule-based systems is event stream processing (ESP) [67, 5, 61, 49, 27, 74]. ESP targets the construction of event-driven information systems, with the aim to extract meaningful patterns from processing event streams. Within the context of rule-based systems, event patterns are defined by domain experts as rules. These rules are executed by the rule engine against streams of events. Once desired event patterns are found, the engine triggers actions according to the rules. Rule-based ESP has found success in many fields such as healthcare, government and other organisations.

The research by Forgy [31] shows that rule engines spend as much as 90 percent of their time performing pattern matching. Over the past few decades, many pattern matching techniques have been developed, such as the RETE algorithm [29] and the TREAT algorithm [59]. These algorithms compile the rules into a network data structure, which consists of several computing nodes representing partial conditions of the rules, in order

to speed up the matching process. The research on rule engines has contributed to different research areas, such as Complex Event Processing, Active Database, and Data Stream Management Systems.

Nowadays, with the development of the Internet of Things (IoT) and the popularity of smart devices, more and more events are generated by a vast variety of sources. The dramatic increase of the volume of event streams brings a big challenge in term of performance to current rule-based systems that are usually built with a centralised architecture. As a result, these systems are inadequate to process larger numbers of rules and bigger data sets we are facing today.

Distribution can improve the performance of rule-based systems. Over the past few years, many approaches for distributing rule engines have been proposed [61, 66, 72, 91, 92]. However, there are some issues in these approaches which limit the potential performance improvement brought by distribution. Among these issues are the workload imbalance and low parallelism due to the way that event streams are processed. More specifically, in these approaches the workload of each computing node in the network is distributed to a statically assigned cluster, in which the events are processed one at a time. Indeed, this way of processing can bring some speed-ups. However, it ignores the inherent workload imbalance among the nodes on the network level. Furthermore, the parallelisation effort is heavily penalised as a result of the one-at-a-time processing model. To address these issues, this thesis proposes a new architecture called DRESS for rule-based systems using techniques from the big data community.

The novelty of this study lies in the construction of the new architecture for processing large event streams. First, although numerous attempts have been made to distribute and parallelise RBSs, the effort usually came from within the RBSs community where a shortage of the expertise for massive data processing can be found. As a result, current distributed RBSs in the market are usually compromised with the aforementioned issues. We argue that we should recognise decoupling as a fundamental value of computer science and, instead of attempting to build their own parallelisation methods, the

RBSs community should embrace the finer big data technologies. Based on this argument, the proposed architecture is built on top of the Spark Streaming framework and adopts some of the advanced parallelisation strategies from the big data community.

Second, despite the adoption of RBSs in event stream processing, RBSs were not designed to process streams. For example, early rule engines usually require the data to be loaded before it is processed. Later development in the research on RBSs, such as the implementation of the RETE and Treat algorithms and their distributed variants, made it possible to process streams with RBSs. However, these algorithms process one event at a time, which is an inherent limit from their design and a big drawback to their performance. This study stands on a stream processing perspective: it does not consider ESP as an application of RBSs instead it views ESP as the main target problem and RBS as a tool to solve this problem. By doing so, we were allowed to remove the inherent limit of RBSs by introducing micro-batching into the proposed architecture. This dramatically improves the performance of RBSs and enables RBSs to process large event streams we are facing today.

1.1 Rule-based Event Stream Processing

Rule-based systems provide a solution for capturing, representing, storing, reasoning about, and applying human knowledge. They automate the process of building expert systems for different areas where job excellence requires consistent reasoning and practical experience [40]. Although artificial intelligence (AI) researchers have developed alternative ways to capture and manipulate knowledge (for example, machine learning models), RBSs have two distinguishable characteristics. First, knowledge is well defined which allows human experts to refine existing knowledge and add new knowledge. Second, RBSs are able to explain their reasoning, making their logic transparent.

The advantages of RBSs can be summarised as follows:

- **Separation of business logic from the processing**: with rule-based systems, the

business logic is stored in the rule base and separated from the processing. This rule base provides a single source of truth (SSOT) for the business logic, enabling different expert systems to be constructed using the same rule engine.

- **Speed and scalability**: many algorithms such as RETE [31], Treat [59] and Leaps [7] provide efficient execution of rules. In addition, with optimisations such as *node sharing* and *parallel rule execution*, rule-based systems are fast.

- **Declarative rule representation**: the rules are usually written in declarative languages which allows the users to tell the system "what to do" instead of "how to do". More specifically, a rule can be written in the form <*IF condition is satisfied THEN what to do*>. The determination of the satisfaction of the condition is carried out by the rule engine and, as a result, the users of RBSs only need to specify the action part (what to do) of the rule.

- **Transparent reasoning**: rule-based systems are able to explain their reasoning and justify their conclusions.

Event stream processing (ESP) targets the tasks of processing event streams with the aim of identifying meaningful patterns within those streams. This concept fits well with rule-based systems. More specifically, human experts can define their desired patterns in a set of rules. Then, these rules are compiled into a network of computing nodes, and the rule engine identifies event patterns by executing the network.

1.2 Statement of the Research Problem

Many approaches have been proposed to improve the performance of rule-based systems [6, 92, 63, 66, 73]. Most of them focus on the parallelisation and distribution of the RETE algorithm, which is the fundamental algorithm behind many rule engines.

For example, the authors of [6, 92] proposed two approaches based on a similar message-passing model in order to distribute the nodes of the RETE networks. In their

approaches, every node of the RETE network is distributed to a cluster of nodes which share the workload of that particular node, with the aim of parallelising the computation. In order to maintain a global state across the clusters these approaches come with a centralised memory, or some kind of synchronisation mechanisms, such as Flux [68].

In [62], the authors proposed a parallel version of RETE based on a method called vector-based matching that can work on GPUs. This approach maintains the global state in the main memory of the GPU and does not support multiple GPUs, which penalises its scalability. Although some improvements have been made in [63], this work does not fit well with event stream processing due to the fact it requires to load the data into the memories of GPUs before processing.

[66] presents an architecture based on WS-Coordination [14], which allows the integration of multiple rule engines into a single system. This architecture requires the decomposition of the rules and its performance heavily relies on how the rules are divided into sub-rules.

The problems of current techniques for distributing RBSs can be characterised as follows:

1. **Static job assignment**: they focus on distributing rule engines at the node level, while ignores the fact that the workloads among the nodes on the network level might be unbalanced.

2. **Centralised memory model**: in order to maintain the global state, they adopt a centralised memory model, which introduces an overhead for transferring data across the clusters.

3. **Low parallelism**: they process one event at a time which penalises the parallelisation effort.

4. **Constant rebalancing**: the workloads across the network tend to change over time, which introduces an extra cost for rebalancing the network.

5. **Lack of fault tolerance**: they neglect the importance of the ability to recover from system failures.

The present thesis addresses the above issue in an investigation of the characteristics of distributed rule based systems and through the proposal of an approach to the distribution of a rule engine.

1.3 Contribution of the Thesis

The main contributions of this thesis can be summarised as follows:

- A distributed rule-based architecture for processing large event streams based on:
    1. a dynamic job assignment model to improve load balance,
    2. the micro-batching technique to increase parallelisation, and,
    3. a decentralised memory model.
- Formalisation of RETE-based rule engines.
- A 2-step verification approach to prove the correctness of transformations for RETE-based systems, which consists of:
    1. the orderless equivalence between two RETE-based systems, and
    2. the order preservation in distributed RETE-based systems.
- An automated transformation from the current RETE-based systems to the proposed model (DRESS), based on MDA.
- A generic benchmark for evaluating the performance of distributed rule engines.
- A case study of large event stream processing with DRESS.

1.4 Publications

During the course of the PhD research, some aspects of the work presented in this thesis have been published as research papers. This thesis provides detailed information for the work presented in the following publications:

[19] Chen, Y. and Bordbar, B. (2016) “Dress: A rule engine on spark for event stream processing.” In Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies. ACM. pp. 46–51

[20] Chen, Y. and Tino, P. (2018) "Formalisation and verification of distributed rule-based expert systems". Paper submitted to journal Expert Systems with Applications.

Chapter 4 of this thesis uses the idea presented in the following work that the author has participated in:

[10] Bowles, J., Alwanain, M., Bordbar, B. and Chen, Y. (2014) “Matching and merging scenarios automatically with alloy.” In International Conference on Model-Driven Engineering and Software Development. Springer. pp. 100–116

1.5 Structure of the Thesis

This thesis is structured as seven chapters including this introduction (**Chapter 1**).

Chapter 2 begins with a background of some of the concepts related to rule-based systems. For example, the rules, events and components of rule-based systems. This is followed by an formalisation of the RETE algorithm. Then, this chapter reviews current approaches for distributed rule-based event stream processing. The review discusses a number of different methods to distribute the rule engines, as well as their benefits and challenges. The objective of this background is to identify the aspects of current approaches that can be improved.

Chapter 3 presents the proposed model for rule-based large event stream processing. It describes the architecture and each component of the model and introduces the strategies used for load-balancing and parallelisation, especially micro-batching and dynamic job assignment. This leads to an distributed version of the RETE algorithm which we call it DRESS.

In **Chapter 4** , an automated transformation from the current RETE based models to the DRESS model is illustrated. This is based on the Model Driven Architecture (MDA) which involves meta-modelling and a set of transformation rules that map the source RETE network to the target DRESS network. This automated transformation is instantiated with the help of the SiTra framework.

In **Chapter 5**, the transformation from RETE to DRESS is verified. The verification consists of two parts. The first part verifies that the DRESS model transformed from a RETE model generates the same set of outputs, if both models are given the same input. And the second part verifies that the DRESS model preserves the order of the outputs of the original RETE model.

Chapter 6 presents a generic and highly configurable benchmark for evaluating distributed rule engines. The main advantage of the proposed benchmark is that it focuses on streaming applications of the engines and it can simulate different real world applications with its configurability. Experiments of DRESS with different configurations of the benchmark were conducted, and the results are discussed.

Chapter 7 concludes this thesis and explores directions for future research.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we introduce the background of rule-based systems. It begins with a formalisation of the RETE algorithm from an event stream processing perspective. Then, current approaches to both distributed rule-based systems and event stream processing systems are investigated with the aim of identifying their advantages and disadvantages. This is followed by an introduction to the methods that are used in the verification chapter (5) and model transformation chapter (4), such as Petri Nets and Model Driven Development.

2.1 Rule-based Systems

In the literature, the terms expert systems, rule-based (expert) systems, knowledge-based (expert) systems, production systems and intelligent systems are used synonymously [43], although some of them, such as knowledge-based systems and intelligent systems, may not be 'rule-based'. To avoid confusion, in this thesis we use the term rule-based (expert) system as it provides the most precise meaning to the describe our work.

A rule-based system (RBS) emulates the decision making ability of human experts using rules written in IF-THEN statements. It works as an observer over the event streams and, when a pre-defined event pattern is observed, it triggers actions according to the rules. Rule-based systems have three major advantages: 1) they are highly accessible

that they fill the gap between the end users and computer technicians by enabling the users to express their desired patterns using rules written in Domain Specific Languages (DSLs), and, 2) they can compile a large number of rules into a graph or network and use optimisation algorithms based on structural similarities to reduce the workload, and, 3) they store partially matched patterns in their working memories to reduce unnecessary computations for future execution cycles.

In the remainder of this section, we discuss the architecture of rule-based systems and formalise the RETE algorithm.

2.1.1 Components of Rule-based Systems

A typical rule-based system consists of three major components:

- **Rule Base** which contains a set of *rules* designed by human experts.
- **Inference Engine** which infers new *facts* based on the *rules* and the existing *facts*.
- **Working Memory (also called Fact Base)** which stores all *facts* of the system.

In addition to the above components, a user interface, which is not a part of the reasoning process, is also essential to applications of rule-based systems. It provides the users with utilities to design and manage the system. Figure 2.1 shows the structure of a typical rule-based system.

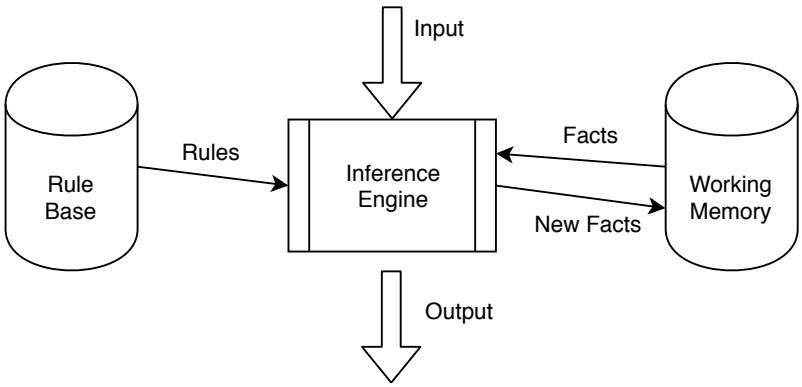


Figure 2.1: Structure of A Rule Based System

Rule Base

The rule base provides a single source of truth (SSOT) for storing the business logic, such that different expert systems can be built by manipulating the rules without altering the implementation of the rule engine. It contains a collection of rules often written in a Domain Specific Language (DSL) and an algorithm is used to process these rules into an intermediate format that the **Inference Engine** can work with. The RETE algorithm [31] is the de facto algorithm to process such rules. A data structure called the RETE network is generated from the rules, which will be elaborated upon in Sections 2.1.4 and 2.1.5.

Inference Engine

In early work, the rules of rule-based systems were intertwined within custom-crafted software. There was an important development in the late 1970's and the early 1980's that several frameworks, such as EMYCIN [77], were constructed to aid in designing new rule-based systems. One of the key ideas behind these frameworks was to separate rules as much as possible from the procedures that manipulate them. As a result of the separation, the design of new systems only focuses on the rules, without considering too much about the coding or performance of the systems [13]. An Inference Engine then applies the rules against the working memory and obtains new knowledge.

According to Friedman [41], a typical inference engine consists of a pattern matcher, a conflict solver and an execution engine. In a processing cycle, the pattern matcher firstly applies the rules to the working memory and finds those rules whose conditions are satisfied by the existing facts. Those rules along with the facts that satisfy their conditions are added into a so-called *conflict set*. Then, since in most systems only one rule may be activated at a time, a conflict solver is used to decide the order that the rules are activated. This process is called *conflict resolution* [22], in which there are many rule ordering strategies [57]; for example linear scanning or ordering according to some preferences. Finally, the execution engine performs the *actions* of the activated rules which may interact with other parts of the system or result in changes to the working memory. The output of a

rule-based system in one execution cycle, which is called the *agenda*, can be seen as an ordered version of the conflict set.

Working Memory (Fact Base)

The working memory stores all knowledge (facts) learned by the system. At the very beginning, the working memory is initialised based on the facts observed from the nature of the system. For example, consider that we have a rule "IF $\langle A \rangle$ THEN $\langle B \rangle$". This rule itself can be viewed as knowledge because it tells the system that **B** relies on **A**. Logically, this knowledge is equivalent to the fact $\neg A \vee B$. Therefore, these facts observed from the rules are added into the working memory. In each of the future execution cycles, new facts from the input, as well as the facts derived from the existing facts and the rules, are inserted into the working memory.

The working memory also support the deletion of facts. When a fact is deleted, the working memory needs to ensure that any other facts that are derived from it are also deleted. As this research targets the large scale pattern matching for RBSs, the deletion of facts are not considered.

2.1.2 Rules and Events

A rule-based system is a program that uses *rules* to reach *conclusions* based on *facts* [40, 12]. During its execution, new *facts* arrive at the system as events. An event is a collection of data describing what have happened. For example, in a banking system, when a customer deposits a cheque, an event could be generated by the system, which contains the value of the cheque, the account number and date etc.

Definition 2.1. *An event $e = (\iota, \mathbb{K}, \mathbb{V}, f)$ is a tuple, where ι is the index of the event, $\mathbb{K}$ is the set of variables carried by e, $\mathbb{V}$ is the universal set of all possible values in the system, and f is the value assignment function $f : \mathbb{K} \to \mathbb{V}$.*

Recently with the development of Internet of Things (IoT) and other technologies

such as self-driving cars, more and more events are generated by a vast variety of sensors and devices. All these events can be captured and put into the input stream of the system. Through time each event is assigned a unique and incremental index by the system - e_i denotes the event with index i. To avoid confusion, we now formally define the term *stream* that is used throughout the thesis.

Definition 2.2. *Unless otherwise specified, a stream $S : \mathbb{N} \to \mathbb{X}$ of a variable x is an injective function where $\mathbb{X}$ is the universal set of all possible values of x and $\mathbb{N}$ is the set of natural numbers. A sub-stream $S\restriction_A : A \to \mathbb{X}$ of a stream S is a restriction of S to A where A is a subset of $\mathbb{N}$. The image of the stream function S is denoted as $img(S)$.*

For example, a stream $S : \mathbb{N} \to \mathbb{E}$ of events indicates that $\mathbb{E}$ is the set of events and S maps each natural number to an event. The input of a rule-based system is a stream $S^{in} : \mathbb{N} \to \mathbb{E}$ defined by $S^{in}(i) = e_i \in \mathbb{E}$, where $\mathbb{E}$ is the set of all events of the system.

To check a particular feature of an event, one can write a propositional statement over the event; for example, 'This is a deposit event' and 'The value of the cheque is £100'. The following notation can be used to assign a propositional statement to a symbol p:

$$p(e) \stackrel{def}{=} \text{'This(e) is a deposit event'}. \tag{2.1}$$

In a rule-based system, a propositional statement over an event $e = (\iota, \mathbb{K}, \mathbb{V}, f)$ is in the following form:

$$f(k) \divideontimes v, \text{ where } k \in \mathbb{K}, \divideontimes \in \bowtie, v \in \mathbb{V},$$

where $\bowtie$ is the set of comparison operators used in rule based systems:

$$\bowtie = \{<, >, \leq, \geq, =, \neq\}.$$

For example, the propositional statement (2.1) can be expressed as:

$$p(e) \stackrel{def}{=} f(type) = deposit,$$

which means "the variable *type* $\in \mathbb{K}$ of the event e has the value *deposit*". We can now formally define a proposition.

Definition 2.3. *Given a stream of events represented by function $S : \mathbb{N} \to \mathbb{E}$, a proposition $p : img(S) \to \{true, false\}$ is a function that maps an event $e \in img(S)$ to either true or false, where $img(S)$ is the image of the function S.*

To reach a conclusion, the rule engine needs to find events (facts) that match some patterns defined by the conditions of rules.

Definition 2.4. *A pattern P (also called a conditional element) consists of zero or more propositions over an event. An event e is said to match a pattern if all propositions of the pattern hold for that event, i.e. $\forall p_i \in P : p_i(e)$.*

It is worth noting that there are cases in which a propositional statement involves more than one event; for example, one such statement can be "the variable x of the event $e_i = (\iota_i, \mathbb{K}_i, \mathbb{V}_i, f_i)$ equals the variable y of the event $e_j = (\iota_j, \mathbb{K}_j, \mathbb{V}_j, f_j)$". In such a case, one pattern for each of the two events are created, along with other propositions for these two events. Then a *variable binding* is used to connect the patterns created for e_i and e_j. The following is a fraction of a rule with a variable binding written in the DRL format (see Section 2.2):

$$\$e_i : Pattern(\ \langle \text{propositions for pattern } e_i \rangle\ ) \text{ and}$$
$$\$e_j : Pattern(\ \langle \text{propositions for pattern } e_j \rangle\ ) \text{ and}$$
$$\$variable\_binding : \langle f_i(x) == f_j(y) \rangle$$

The processing of the *variable binding* will need to ensure that the variable y is bound to the same value of variable x, which is from another event.

Rules

In rule-based systems, the design of rules (sometimes called production rules) focuses on the representation of knowledge in order to express propositional and first order logic in

a concise and declarative manner. These rules are usually written as IF-THEN statements in the following format:

$$\textit{rule}\text{: IF } \langle \textit{condition} \rangle \text{ THEN } \langle \textit{action} \rangle.$$

Remark. *A rule can also be expressed by its condition on the left-hand-side (LHS) and its action on the right-hand-side (RHS):*

$$\textit{LHS(rule)} = \textit{condition} \qquad \textit{RHS(rule)} = \textit{action.}$$

A rule is said to be activatable (or fireable) if its *condition* is fulfilled. The *condition* of a rule is a logical compound built up of one or more *patterns*. The most basic form of a *condition* is a conjunctive clause of patterns as follows [55]:

$$P_1 \wedge P_2 \wedge \cdots \wedge P_n, \tag{2.2}$$

where the patterns must be matched simultaneously in order to fire the rule.

Let h be the *action* of a rule, we can rewrite the rule as follows:

$$\textit{rule} : P_1 \wedge P_2 \wedge \cdots \wedge P_n \rightarrow h. \tag{2.3}$$

Definition 2.5. *An atomic condition $c = P_1 \wedge P_2 \wedge \cdots \wedge P_n$ is a logical expression that contains n patterns $P_i, 1 \leq i \leq n$, and these patterns are connected only by conjunctions($\wedge$). An atomic rule is a rule whose condition is atomic.*

A more complex *condition* Φ can be an arbitrarily formula of propositional logic (with both $\wedge$ and $\vee$ connectives). In such a case, this complex *condition* can be transformed to its Disjunctive Normal Form (DNF)(2.4) using logical equivalence laws, such as the double negative elimination, De Morgan's laws, and the distributive law [25],

$$\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n \rightarrow h, \tag{2.4}$$

where each ϕ_i is an atomic condition (conjunctive clause of patterns) as in (2.2). Consequently, a rule $rule : \Phi \rightarrow h$ with a complex condition can be replaced with n atomic rules:

$$rule_1 : \phi_1 \rightarrow h,$$
$$rule_2 : \phi_2 \rightarrow h,$$
$$\ldots$$
$$rule_n : \phi_n \rightarrow h.$$

Each atomic rule will be compiled into a RETE network, as explained in the following sections.

Facts

If we consider $\rightarrow$ as the equivalent of $\Rightarrow$ (implication), the rule in (2.3) is logically equivalent to:

$$\neg(P_1 \wedge P_2 \wedge \cdots \wedge P_n) \vee h. \tag{2.5}$$

The above is the format of *facts* in the *fact base*. The *fact base* is the memory which stores all knowledge a RETE-based system has gathered. At the beginning, the *rule base* is initialised with *facts* in above format for all rules in the *rule base*. The arrival of an event may change a pattern P_i from unmatched to matched. As a result, P_i is added to the *fact base* as a *fact*. Some intermediate *facts* may also be observed from existing *facts*. For example, if in the *fact base* we have P_1 and $\neg(P_1 \wedge P_2 \wedge P_3) \vee h$, since P_1 is already matched, an intermediate *fact* $\neg(P_2 \wedge P_3) \vee h$ may also be added into the *fact base*.

2.1.3 Forward and Backward Chaining

For an inference engine, there are two common strategies for deriving new facts from the rules and the existing facts: forward chaining and backward chaining [3, 69, 71].

Forward chaining is like a breadth-first search algorithm. It begins from the existing facts and derives new facts according to the rules. Backward chaining, on the other hand, is like a depth-first search algorithm. It begins with a desired goal and goes backward to see if the existing facts support the goal. In this section, we illustrate the processes of both strategies in an pattern matching example with three rules and three initial facts, as shown in Figure 2.2.

Initial Facts : $A\ B\ D$

Rule 1 : $A \rightarrow C$

Rule 2 : $B \wedge C \rightarrow E$

Rule 3 : $D \wedge E \rightarrow F$

Figure 2.2: Example Problem for Chaining Strategies

Forward Chaining

Forward chaining is also known as a data-driven inference technique. It examines the set of rules and infers new facts based on these rules. In each execution cycle, the engine applies the rules to the working memory until a conclusion is reached or no other rules can be applied. If a conclusion is reached, new facts are added to the working memory according to the RHS of the activated rule.

Consider the example in Figure 2.2. In the first cycle, the engine applies rule 1 to the working memory and derives C as a result of the conclusion of the rule. Hence, C is added into the working memory. Then, in the next cycles, E from the conclusion of rule 2 and F from the conclusion of rule 3 are also added to the working memory. Figure 2.3 illustrates the process.

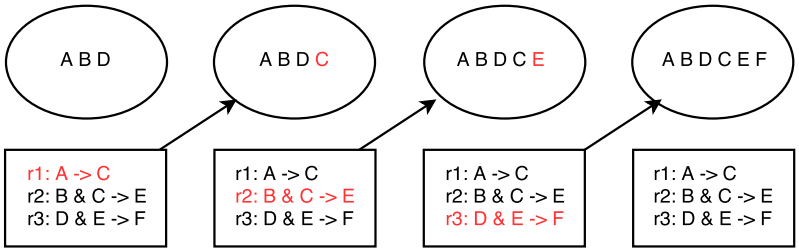


Figure 2.3: Forward Chaining Strategy

Forward chaining is a popular implementation strategy for rule-based expert systems. For instance, the RETE algorithm implements forward chaining by storing partially matched patterns as facts. Consider the rule $B \wedge C \rightarrow E$ of the above example. RETE analyses this rule and known facts (A, B and D) and a partially matched pattern $C \rightarrow E$ is stored in its working memory because B is a known fact. It is important to note that each instance (event) of B will contribute to an instance of the partially matched pattern $C \rightarrow E$. As the number of rules and the size of dataset grow, the working memory store a significant number of items. Hence, this strategy can be seen as a space–time trade-off.

In distributed rule-based systems, the working memory can be implemented either as a centralised memory, or as a distributed memory or database. The centralised memory model, as adopted by [92], is highly maintainable but at a cost of data transfer latency. On the other hand, [91] adopts the distributed memory model, which has to implement a complex mechanism to guarantee state consistency.

Backward Chaining

Backward chaining is known as a goal-driven inference technique. It starts from a possible goal (conclusion) and examines the working memory for facts that can satisfy the conditions of that goal.

Consider the rule $D \wedge E \rightarrow F$ of the example in Figure 2.2, in which F is the goal. The engine examines the working memory for the facts to satisfy the conditions (D and E). Since E is not in the working memory, the engine creates an antecedent goal, based on rule 2, to find E. Next, since rule 2 needs C, another antecedent goal is created to find C.

Finally, from rule 1, C can be derived. The engine then goes backward to process the list of goals until F is found. This process is shown in Figure 2.4.

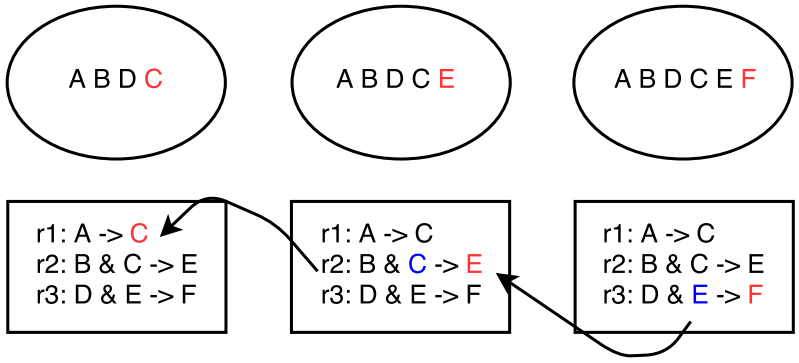


Figure 2.4: Backward Chaining Strategy

Backward chaining is useful if there is a need to query the working memory during the execution of the system, i.e. to add new rules to system during its execution. Many of the current rule engines, such as Drools[67] and JESS[41], support both forward and backward chaining.

2.1.4 The RETE Algorithm

The RETE (Latin for net) Algorithm is a forward chaining pattern matching algorithm designed by Dr Charles L. Forgy in the late 1970's [29], and it is commonly used in implementing rule-based systems such as Drools [67] and JESS [41]. Furthermore, it is behind many rule-based programming languages such as CLIPS [34] and OPS5 [30]. One of the most important jobs for rule engines is pattern matching, during which the rule engine examines each rule and searches the fact base to determine whether the rule's conditions have been satisfied. If the conditions of a rule are satisfied by existing facts, it is added into the conflict set. Then, a conflict solver is used to decide the order of activating the rules in the conflict set. Afterwards, the activated rules are added into the *agenda*. Figure 2.5 shows this process.

The major disadvantage of having the rule engine check each rule to direct the search for facts in the whole fact base, as explained by Giarratano [34], is that it can be very

slow. Typically, in each execution cycle only a few facts are changed, which makes up only a small fraction of the fact base. As a result, having the rule engine to check the entire fact base for each rule requires a lot of unnecessary computation, as most of the rules will likely find the same facts as they did in previous cycles.

The RETE algorithm removes this disadvantage by saving partially matched patterns from cycle to cycle into a data structure called the RETE network. This is a tree-like structure where each branch represents a rule, and each node in the network represents a computing unit that checks a partial condition of the rule. In a cycle, the RETE algorithm only recomputes facts that have changed.

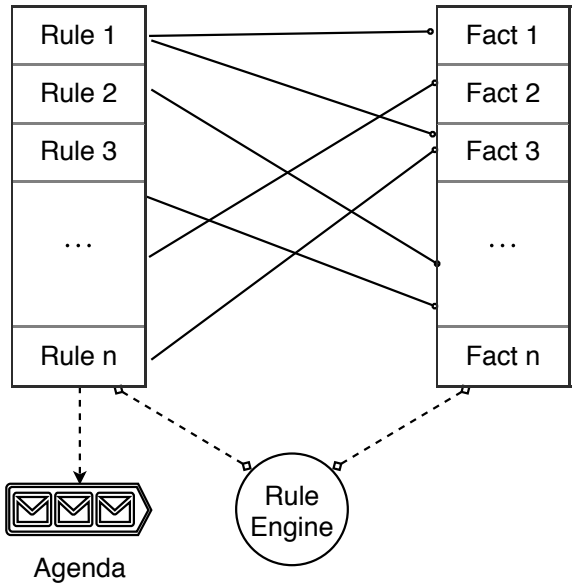


Figure 2.5: Pattern Matching Process of Rule Engines

2.1.5 Construction of RETE Networks

The RETE algorithm has been commonly used in implementing rule based systems [38, 48, 6]. A data structure called the RETE network is generated by the algorithm from a set of rules. Typically, this network consists of four types of nodes:

- **Root Node**: the single node where new facts (events) enter the network.
- **Alpha Node**: the nodes responsible for selecting events based on simple conditional

tests.

- **Beta Node**: the nodes responsible for joining two nodes.
- **Terminal Node**: the special beta nodes representing activations of rules.

Regardless of implementation methods, RETE networks have the following properties:

- A RETE network is a directed acyclic graph that represents higher-level rules.
- The input of the RETE network is a stream of events. These events carry some data and are ordered by some kind of indices.
- Alpha nodes are usually on the top of RETE networks. They accept either the input stream of the system or the output of another alpha node. An alpha node works as a filter such that its output stream consists of all events from its input stream that satisfy its corresponding proposition.
- Each beta node aggregates the output streams of two nodes to create a new stream containing tuples of items from both streams that satisfy the condition associated with that beta node.
- The output streams of beta nodes at the bottom of the network are not processed further. These beta nodes are called terminal nodes and their output streams form the output of the RETE network.

Figure 2.6 shows an example of a RETE network.

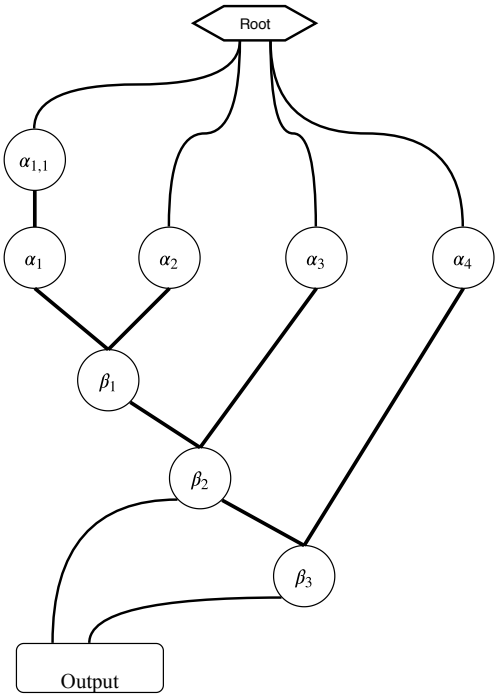


Figure 2.6: Example of A RETE Network

Root Node

New facts (events) enter a RETE network at its root node. The output of the root node is a stream of events and it is broadcast to alpha nodes. Visually, a root node is represented by a hexagon, as shown in Figure 2.7.

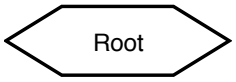


Figure 2.7: RETE - Root Node

The output of the root node can be seen as a stream of events represented by function $S^{in} : \mathbb{N} \rightarrow \mathbb{E}$.

Alpha Node

Consider a pattern P. The propositions p_i in P test different attributes of a given event. The RETE algorithm creates an alpha node α_{p_i} for each proposition p_i and connects the created alpha nodes into an *alpha chain*. Upon the arrival of an event, the first alpha node

in the alpha chain is activated to test for the presence of a particular attribute of the event, as defined by its corresponding proposition. If the test succeeds, the event is sent out to the next alpha node in the chain for testing other attributes. If the test fails, the event is simply dropped by the node.

Consequently, an atomic rule $r = P_1 \wedge P_2 \wedge \cdots \wedge P_n \rightarrow h$ will result in a set of alpha chains. The output of the root node is broadcast to the first node of each alpha chain. Moreover, each alpha chain terminates at a working memory ω^{α} called the *alpha memory*, which stores the output of the last node of the chain. All alpha nodes and alpha memories form an *alpha network*, as shown in Figure 2.8

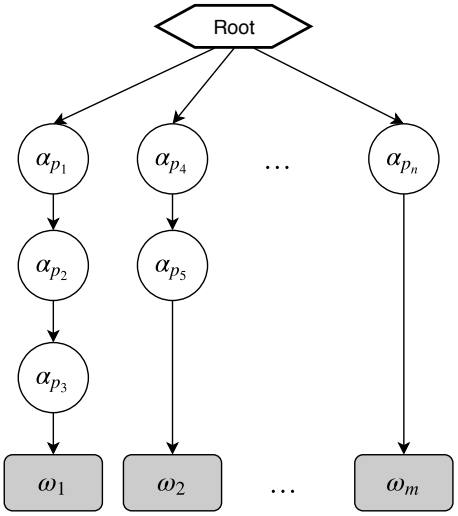


Figure 2.8: RETE - Alpha Network

Mathematically, an alpha node α_{p_i} corresponding to proposition p_i could be seen as a restriction to the function representing the input stream of α_{p_i}. Given an index subset $A \subseteq \mathbb{N}$, corresponding to the input stream $S\restriction_A : A \rightarrow \mathbb{E}$ of an alpha node α_{p_i}, α_{p_i} restricts the domain A to A_{p_i}, which is the set of indices j from A such that $p_i(e_j)$ is true. In other words, $j \in A_{p_i} \iff p_i(e_j) \wedge (e_j = S\restriction_A(j))$. Consequently, the output of α_{p_i} can be seen as the function:

$$S\restriction_{A_{p_i}} : A_{p_i} \rightarrow \mathbb{E}, \text{where } A_{p_i} \subseteq A.$$

The function $S|_{A_{p_i}}$ is called the alpha function, corresponding to the alpha node α_{p_i}.

The output of an alpha node can be consumed by other alpha nodes in the same alpha chain. Hence, for each alpha chain, a series of alpha functions is created:

$$(S\restriction_A)\restriction_{A_{p_1}} : A_{p_1} \to E, \text{where } A_{p_1} \subseteq A$$
$$((S\restriction_A)\restriction_{A_{p_1}})\restriction_{A_{p_2}} : A_{p_2} \to E, \text{where } A_{p_2} \subseteq A_{p_1}$$
$$\dots$$
$$((((S\restriction_A)\restriction_{A_{p_1}})\restriction_{A_{p_2}})\dots)\restriction_{A_{p_n}} : A_{p_n} \to E, \text{where } A_{p_n} \subseteq A_{p_{n-1}}$$

To ease mathematical notation we will denote $(((S\restriction_A)\restriction_{A_{p_1}})\restriction_{A_{p_2}})\dots)\restriction_{A_{p_n}}$ by $S\restriction_{A_{p_1,p_2,\dots,p_n}}$. By combining all functions of an alpha chain $\alpha_{p_1,p_2,\dots,p_n}$, we have the domain $A_{p_1,p_2,\dots,p_n}$ of the output function of the last node α_{p_n}, satisfying $j \in A_{p_1,p_2,\dots,p_n} \iff \bigwedge_{i=1}^{n} p_i(e_j) \wedge (e_j = S|_A(j))$, where $S|_A$ is the stream function representing the input of the first node in the chain. Then, the alpha chain $\alpha_{p_1,p_2,\dots,p_n}$ can be seen as the representation of the pattern (a logical conjunction of propositions) over the same event:

$$S\restriction_{A_{p_1,p_2,\dots,p_n}} : A_{p_1,p_2,\dots,p_n} \to E.$$

Finally, the output of the above function is stored at the alpha memory of the alpha chain.

Beta Node

Beta nodes perform joins between beta memories and alpha memories. A beta node is also called a 2-input node, which has a *left* and a *right* input. Consider a set $W_\alpha = \{\omega_1^\alpha, \omega_2^\alpha, \dots, \omega_m^\alpha\}$ of alpha memories. The first alpha memory $\omega_1^\alpha \in W_\alpha$ is directly adapted to a beta memory ω_0^β without processing. For any other alpha memory ω_i^α, a beta node β_{i-1} is created such that its *left* input is the beta memory ω_{i-2}^β, and its *right* input is the alpha memory ω_i^α. The beta node then joins the two working memories and creates a stream of tuples of elements from the memories. This stream is stored at a beta memory ω_i^β. All beta nodes and beta memories form the *beta network*, as shown in Figure 2.9.

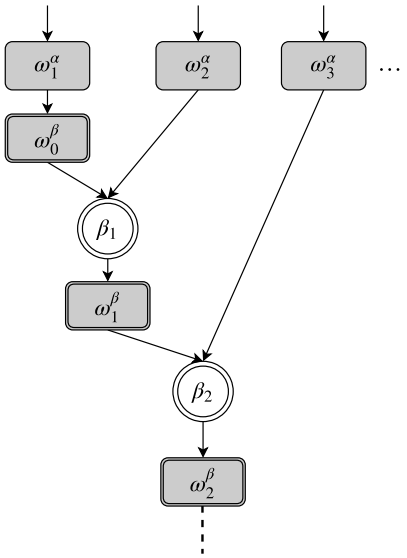


Figure 2.9: RETE - Beta Network

A beta memory can be represented by a stream $S\colon \mathbb{N} \to \mathbb{T}$, where $\mathbb{T}$ is the set of all possible tuples of events. As mentioned above, the first alpha memory ω_1^α of an alpha network is adapted to a beta memory ω_0^β without processing. Given the alpha function $S\restriction_{A_{p_1}} : A_{p_1} \to \mathbb{E}$ corresponding to ω_1^α, the first beta memory ω_0^β is represented by the function $S^{\beta_0} : \mathbb{N} \to \mathbb{T}$. The image of the function S^{β_0} is a set of 1-tuples:

$$S^{\beta_0}(\mathbb{N}) = \{(e_i) \mid e_i = S\restriction_{A_{p_1}}(i)\}$$

Mathematically, a beta node is a function that maps two working memories to a beta memory. Consider a beta node β_{i+1}. Given its left input - a beta memory ω_i^β represented by function $S^{\beta_i} : \mathbb{N} \to \mathbb{T}$, and its right input - an alpha memory ω_j^α represented by $S\restriction_{A_{p_j}} : A_{p_j} \to \mathbb{E}$, the beta node β_{i+1} joins the memories and produces a stream of tuples of working memory elements (WMEs):

$$S^{\beta_{i+1}} : \mathbb{N} \to img(S^{\beta_i}) \times img(S\restriction_{A_{p_j}}).$$

The stream is then stored at a beta memory ω_{i+1}^β. Beta nodes that can be presented by the above function are called *join* nodes.

Consider a rule with one or more variable bindings (refer to Section 2.1.2, page 14).

Each variable binding works as an additional condition for a tuple to be inserted into a beta memory. Let $\bowtie$ be the operator to test the variable binding of two WMEs, the output stream of the beta node with a variable binding can be represented by:

$$S^{\beta} : \mathbb{N} \rightarrow \{(x,y) \mid x \in img(S^{\beta_i}) \wedge y \in img(S \restriction_{A_{p_j}}) \wedge x \bowtie y\}.$$

The beta nodes that can be presented by above function are called *variable binding* nodes.

2.1.6 Optimisation of RETE Networks

The RETE algorithm allows an important optimisation technique called *node sharing*, which takes the advantage of the structural similarities of the RETE networks. Node sharing is based on the fact that many rules may share a same part of propositions. As a result, some of the nodes among the networks will do exactly the same job. In order to reduce the workload, these nodes can be shared by different rules.

For example, consider two atomic rules $r_1 : P_1 \wedge P_2 \wedge P_3 \rightarrow h_1$ and $r_2 : P_1 \wedge P_2 \wedge P_4 \rightarrow h_2$, where each pattern P_i has one and only one proposition p_i. We can create two RETE networks for the rules as shown in part I and II of Figure 2.10. Because the first two patterns P_1 and P_2 are shared by both rules r_1 and r_2, there are duplicated alpha nodes for propositions p_1 and p_2, and duplicated beta nodes for the partial condition $p_1 \wedge p_2$. With node sharing optimisation, duplicated nodes are shared among the network. Hence, these two RETE network are merged into one network, as shown in part III of Figure 2.10.

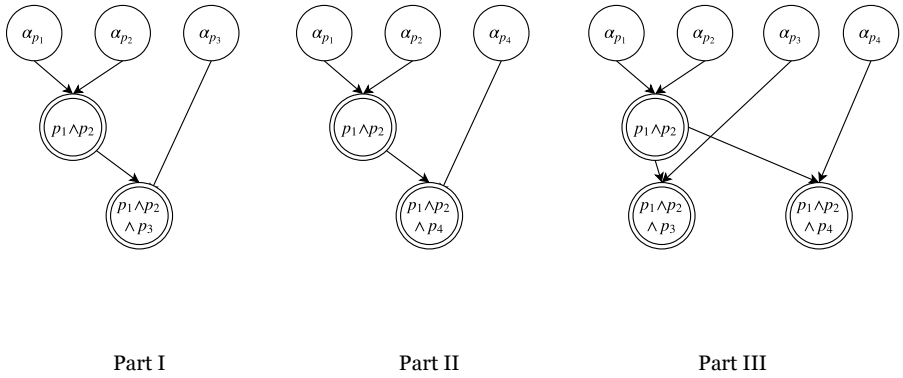


Figure 2.10: Node Sharing Optimisation for RETE Networks

2.2 An Overview of the Drools Business Rules Management System

Drools [67] is one of the most popular Business Rule Management Systems (BRMSs), allowing fast and reliable evaluation of business rules and complex event processing. Drools consists of five components:

- **Drools Guvnor**: a business rules manager providing a single source of truth (SSOT).
- **Drools Expert**: a rule engine which implements and extends the RETE/RETE-OO algorithm.
- **Drools Flow**: provides workflow or business process capabilities to Drools.
- **Drools Fusion**: provides event stream processing capabilities to Drools.
- **Drools Planner**: provides a constraint solver to many optimisation problems.

With Drools, the business logic of the systems is declared as a set of rules, written in the Drools Rule Language (DRL).

A rule written in DRL has four parts:

- **name**: which defines the logical name for this rule.

- **attributes**: which provides a declarative way to influence the behaviour of the rule. For example, the value of the attribute *salience* decides the order of execution when multiple rules are activable, which is used in the conflict resolution process.
- **conditions**: which defines the conditions of the rule.
- **actions**: which defines what actions will be taken when the rule is activated.

The most important part of the rules is the conditions. Each pattern of an atomic rule $r = \{P_1, P_2, \ldots, P_n, h\}$ defined in Section 2.1.2 can be represented by a conditional element in DRL. A conditional element contains the type and zero or more propositions (constraints) of the events. For example, the rule r can be written in DRL as follows:

```
rule "r"
when
    $P1 : EventType( p1, p2, ..., pn )
    $P2 : EventType( <propositions for P2> )
    ...
    and $Pn : EventType( <propositions for Pn> )
then
    <action h>
end
```

Note that the logical conjunction (*and*) can be implicit (it is omitted for P_2 and explicitly expressed for P_n).

The application of Drools consists of two parts. The first part is called *Authoring*, in which a set of DRL rules are created and compiled into an enhanced RETE network. The second part is the *Runtime*, in which the rule engine creates the working memory, loads the data and executes the RETE network. If the rule engine finds rules whose conditions are satisfied during the execution, it triggers actions corresponding to those rules.

The performance of Drools is compared to our model in the experiments presented in Chapter 6.

2.3 Distributing Rule-based Systems

One of the major efforts that has been made to improve the performance of RETE-based rule engines is the parallelisation of the computing nodes of the network [63].

Zhou et al. proposed a rule engine system called RUNES II [92], based on a message passing model. In their work, the RETE network is divided into subnets. These subnets, which communicate with each other through a messaging system, are distributed to a cluster. The performance of this system relies on the decomposition of the RETE network and the overhead of communication across the subnets.

Zhu et al. proposed a MapReduce-based architecture for distributing the RETE networks [93]. In this architecture, the RETE network is decomposed into subnets such that each subnet has at most one beta node. The partially matched condition is then added into the system as an event. This introduces unnecessary computation and penalises the pattern matching performance of the RETE algorithm.

Stephen et al. proposed a distributed rule evaluation and event management system called DRES [72]. DRES distributes instances of its RETE-based rule engine across a cluster of computing nodes. All of these nodes are capable of executing filter operations, which means the workload of the alpha network is evenly distributed. However, only a set of statically assigned nodes act as the beta nodes which execute join operations. As a result, for the applications that require heavy join computation, the performance improvement is limited.

In summary, the current approaches adopts the *static job assignment* strategy to organise the computing resources. *Static job assignment* refers to the distribution method in which the workload of the original computing node is statically assigned to a set of computing resources. It brings two major benefits - **speed**: the workload of the original node is shared by a cluster of nodes; and, **scaling**: the system is scalable, due to the fact that more computing resources can be added.

These approaches transform each node of the network to a cluster of nodes and distribute the workload of that node to the created cluster. Consider the RETE network in

Figure 2.11. The alpha nodes and the beta nodes are transformed to the alpha clusters and the beta clusters, respectively. Moreover, all nodes within a cluster share the workload of the original node, which speeds up the processing as a result of the node-to-cluster distribution.

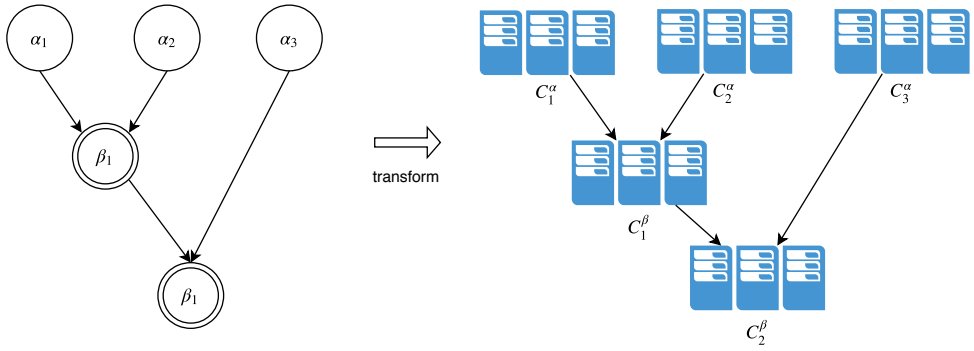


Figure 2.11: Static Job Assignment for Rule Engines

Nevertheless, despite the speed-up brought by the distribution at the node level, there is a major issue which limits the potential of the performance improvement: in the original RETE network, the workloads across the nodes may be unbalanced, which means some nodes may have significantly more work to do than others, while some nodes may be idle for most of the execution time. With the static job assignment strategy, this imbalance is brought to the distributed network. Therefore, if the clusters are created in a fixed or equal size during the distribution, the clusters which have higher workload may become extremely slow. Moreover, because the computation on each node in the RETE networks usually relies on the results of other nodes, these slow clusters will become bottlenecks of the entire system.

Although live reconfiguration of the number of nodes for each cluster can solve aforementioned issue, it comes with a cost. More specifically, the workloads across the network tend to change over time which may result in constant reconfigurations and rebalancing. As the network grows larger, the cost of constant reconfigurations becomes unacceptable.

2.4 Event Stream Processing

Event stream processing (ESP) has been a well studied topic in the field of information systems. Various techniques have been developed regarding ESP, for example, Complex Event Processing (CEP), Data Management Systems (DBMSs), Active Databases, and Rule-based Systems (RBSs). With the increase of data volume over the last decade, ESP has been moving from these approaches to a more generic approach as a big data technology [33]. In the big data community, numerous types of systems have been developed for data processing, and these can be categorised into two groups: batch and stream processing.

The main focus of the batch processing approach is the processing of a large volume of data at once. As a result, this technique requires the data to be accessible before it is processed. One of the keystones in the research regarding batch processing is the release of Google's MapReduce paper [26]. The key idea behind MapReduce is the split-apply-combine strategy [86]. In particular, the map step of MapReduce corresponds to *split* and *apply*, in which the data set is divided into batches and processed in a highly parallel environment. Furthermore, the reduce step corresponds to *combine*, in which the intermediate results obtained from processing the batches are combined. MapReduce has inspired the development of Apache Hadoop [70], which later became one of the most popular big data technologies. Batch processing is an extremely efficient way to process a large amount of data that is collected over a period of time, as it can usually be done simultaneously.

Stream processing approaches, on the other hand, focus on the ability to instantaneously process data streams. In addition, there is no limitation to how long the system will operate. Stream processing is extremely beneficial if there is a need for the events to be processed in real-time and reacted to quickly. One representative of stream processing systems is Apache Storm [75]. Storm is a distributed general-purpose real-time computation system for processing unbounded streams of data. In order to achieve the lowest latency possible, each event in Storm is sent out for processing when it arrives.

Most of the current stream processing approaches [44, 4, 75, 21] adopt the *continuous operator model*, which requires continuous computation as data flows through the system. The *continuous operator model* refers to a way of processing events in which the computing nodes receive one event, update their local state and forward the results to other nodes. This model is subject to the high cost of maintaining a global state for stateful computations, as the processing of an event may rely on the processing of another event.

In addition to speed, fault-tolerance is another important property for both batch and stream processing systems. There are three widely used strategies for recovering from failures:

- **Active replication** [36]: the same event is processing on multiple replications of a computing node. When a failure occurs at a node, the system simply switches to a replication of it. This strategy provides a fast way to recover from failures at a cost of at least twice more computing resources.

- **Passive replication** [52]: snapshots (sometimes called checkpoints) for each computing node are stored at one or more backup nodes. When a failure occurs, the system reconstructs the computing node from these stored snapshots. Different techniques can be used to implement the backup nodes, such as a distributed storage or transactional memories.

- **Upstream backup** [44]: each (upstream) computing node buffers its results until an acknowledgement is received from the downstream nodes. When there is a failure in a particular node, the system reconstructs that node and makes the upstream nodes to resend their buffered results.

None of these strategies look promising in larger clusters. First, the replication strategies (active and passive) require at least double the computing resources. Beyond that, even if the additional resources are affordable, they may still not work when the replicated nodes fail at the same time. Secondly, upstream backup takes a long time to recover

a particular node and the rest of the system has to wait for the recovery to be completed.

The benefits and drawbacks of batch and stream processing can be summarised as follows:

- Batch processing is suitable for applications where having up-to-date data is not important as it processes data with a delay. The advantage of batch processing methods is that they have higher throughput and are more robust than stream processing methods that feature a one-at-a-time strategy.

- Stream processing provides the lowest possible latency as events are processed as soon as they arrive. However, it is difficult to efficiently maintain the processing state and guarantee the high-level fault-tolerance.

ESP applications, in general, have similar requirements to the stream processing model. For example, the system should be able to process data on the fly, and the response time of the system should be fast enough so that the events are reacted to in a timely manner. As a result of these similarities, many distributed ESP systems have been built with a stream processing model.

In order to have the advantages of both batch and stream processing methods, the concept of micro-batching has recently arisen as a hybrid approach to data stream processing [15, 89, 90]. Micro-batching can be seen as a stream processing method but instead of processing one event at a time, it processes small batches of events that are created at regular time intervals (usually in sub-seconds). Micro-batching benefits from the advantages of batching processing (e.g. higher throughput and fault-tolerance) while at the same time keeping the latency as minimal as possible. Apache Spark Streaming is an example of systems based on this micro-batching paradigm. Spark Streaming introduced two abstractions on the data it processes: Resilient Distributed Datasets (RDDs) [88] and Discretised Streams (DStreams) [89]. An RDD is an immutable distributed collection of data (batch) and a DStream can be seen as a stream of RDDs. Spark's unified execution engine for both batch and streaming brings some unique benefits over traditional stream

processing systems. In particular, two major benefits are 1) fast recovery from failures and stragglers, and, 2) better load balancing and resource usage.

To build a rule engine as an event stream processing application, several factors need to be considered. First, the rule engine has to be fast, i.e. it reacts to events in a timely manner. Second, it needs to be able to recover from system failures quickly. Evidently, batch processing methods are not suitable for implementing rule engines, as they process data with a delay. In addition, the stream processing model, which is the dominant paradigm used in many distributed rule engines, has some issues too. For example, its one-at-a-time model results in lower throughput and difficulties to recover from failures. These issues make the micro-batching paradigm a good choice to implement rule engines. Indeed, micro-batching provides high-level fault-tolerance, high throughput and better resource utilisation, which, as a whole, are not provided by either batch or stream processing methods.

2.4.1 An Overview of the Apache Spark Streaming Framework

MapReduce has been highly successful in implementing data sensitive applications. Most of these applications are built around an acyclic data flow model in which the data flows along the system unidirectionally without being reused. On the contrary, Apache Spark [90] focuses on the applications where the data will be reused across multiple parallel operations. It comes with a data abstraction called the resilient distributed datasets (RDDs). An RDD is a read-only collection of data partitioned across a cluster. Therefore, the processing of RDDs can be seen as multiple parallel transformations from these RDDs to other. Spark maintains the *lineage*, which is the history of the transformations, of each RDD. Hence, as long as the original data is in reliable storage, RDDs can always be reconstructed by using its lineage. In addition, as RDDs are partitioned across the cluster, this reconstruction is also done in parallel. This mechanism provides Spark fault tolerance without replication.

Spark Streaming is an extension to Spark. It is a general purpose streaming system

providing near real-time processing of data streams. With Spark Streaming, complex algorithms can be created with high-level parallel operations such as map, reduce, join and window. Data streams can be ingested from many sources, such as Kafka, Flume and ZeroMQ, and pushed out to filesystems, databases, and live dashboards. It supports fault tolerance with the assurance that any specific event is processed exactly once, even with a node failure [89].

Unlike the traditional continuous operator model, where the computation is statically allocated to a node, Spark Streaming discretises the streaming data into micro batches (as RDDs) that can be processed by any node of the cluster. The streams of RDDs, known as the Discretised Streams (DStreams), structures computations as a set of short, stateless, deterministic tasks. These tasks are execute by Spark's batch processing engine, which provides high-level fault tolerance and high throughput.

In addition, as it is backed by a batch processing engine, Spark Streaming can work with resource management systems such as Apache Hadoop YARN [78] and Mesos [42]. These systems enable Spark to reduce the overhead of transferring data by enforcing data locality [39].

2.5 Petri Nets

Petri nets are a graphical and mathematical tool specialised for analysing information processing systems that are characterised as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic [60]. Generally, a system can be modelled by a Petri net with three finite sets of elements: *places*, *transitions* and *arcs* [64]. *Spaces* represent properties of the system and a *transition* represents an action within the system that may trigger changes of the properties. Visually, *spaces* and *transition* are represented by circles and bars respectively. These *places* and *transitions* are connected by directed *arcs*.

Figure 2.12 illustrates an example of Petri net describing a system used in a diagnosis

room at a hospital.

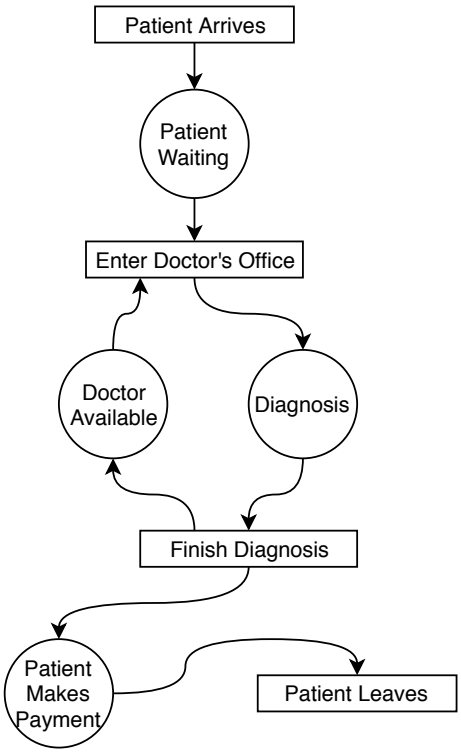


Figure 2.12: Petri Net For A Diagnosis Room System At A Hospital

Definition 2.6. *A Petri net (PN), as adapted from [65], is a tuple*

$$PN = (P, T, Pre, Post),$$

where P is a set of places; T is a set of transitions; $Pre : P \times T \to \mathbb{N}$ is the pre-condition function and $Post : T \times P \to \mathbb{N}$ is the post-condition function.

$Pre(p_i, t_j) = 0$ indicates there is no arc from place p_i to transition t_j, and any positive values of $Pre(p_i, t_j)$ indicates the weight of the arc from p_i to t_j. Analogically, $Post(t_j, p_i)$ indicates the weight of the arc from transition t_j to place p_i.

The dynamic behaviours of systems are modelled by positioning and moving of *tokens* in Petri nets. Tokens (denoted by $\bullet$) reside in places and can be moved around the net by firing transitions. A *marking* $M : P \to \mathbb{N}$ is a function that assigns a non-negative number of tokens to each place of the net. A transition t is enabled (or fireable) if for all places

$p \in P$, $M(p) \geq Pre(p,t)$. The firing of a transition t removes $n = Pre(p,t)$ tokens from places $p \in P$ and adds $n' = Post(t,p')$ tokens to places $p' \in P$.

Figure 2.13 illustrates a transition firing in Petri nets. On the static view, we have $Pre(p_1,t_1) = 2$, $Pre(p_2,t_1) = 1$ defining weights of the arcs from places p_1 and p_2 to transition t_1. Moreover, the arc defined by $Post(t_1,p_3) = 3$ has a weight of 3. On the dynamic view, we have a marking M such that $M(p_1) = 3$, $M(p_2) = 2$ and $M(p_3) = 0$. Transition t_1 is enabled since $M(p_1) \geq Pre(p_1,t_1)$ and $M(p_2) \geq Pre(p_2t_1)$. By firing t_1, corresponding tokens will be removed from p_1 and p_2, and tokens will be added to p_3, yielding a new marking M' as described by the right side of the arrow.

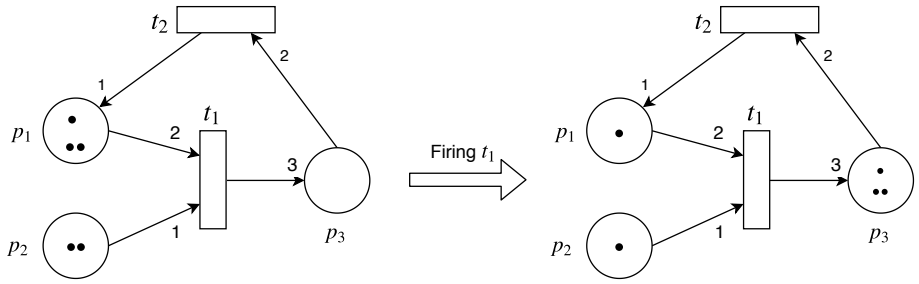


Figure 2.13: Firing of Transitions in Petri Nets

A marking M' is reachable from a marking M on a Petri net if, after firing a sequence of transitions, the Petri net starts from marking M and ends up at marking M'. Given an initial marking M_0 for a Petri net PN, we can draw a graph of all reachable markings from M_0 connected by arcs representing transitions. This graph is called the *reachability graph* for RN.

A reachability graph is a directed graph. The nodes of a reachability graph correspond to markings reachable from the initial marking, one node per marking. The arcs are one-step transitions from one marking to another. Figure 2.14 shows an example of a reachability graph.

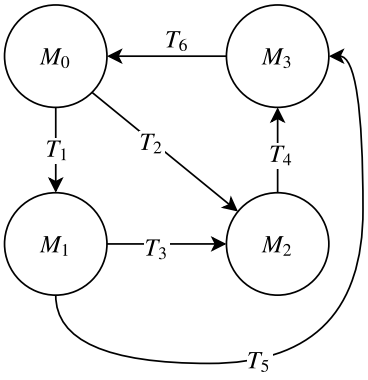


Figure 2.14: Example of Reachability Graph

2.6 Model Driven Development (MDD)

The application of modelling in software development has a long history and has become popular since the development of the Unified Modelling Language (UML) [76]. The modelling in a common model-based development, however, is merely intentional because after all the implementation relies on the programmer's interpretation of the models and realisation of the ideas into code. This is problematic because software systems are liable to changes and those changes can only result in complex model-adapting tasks or inconsistency between models and implementations.

Model-Driven Development (MDD) [80] takes a different approach: it emphasises the analysis of systems and the design of models in software development, and considers models as an equivalent to the code. The implementation is then automated by code generation techniques [85, 17]. Hence, MDD reduces the development life cycles. Another advantage brought by MDD is that different platform-specific implementations can be generated from the same set of models, because the design of models is separated from the implementation.

It is generally agreed that meta-modelling is an essential foundation of MDD [35, 32]. A meta-model describes the minimum set of elements required in order to model a system, and the concrete system model is an instance of the meta-model. In MDD, a model transformation can be defined as mapping the meta-elements: all elements and

the interactions between elements are mapped from the source meta-model to the target meta-model. This can be achieved by a model transformation framework with a set of transformation rules. Subsequently, models derived from the source meta-model can be transformed automatically to an instance of the target meta-model.

Figure 2.15 illustrates the outline of model transformations in MDD. The model transformation framework takes the source and target meta-models, as well as the transformation rules as inputs. For any instances of the source meta-model, the framework creates a target model according to the meta-models and the rules describing how elements of the meta-models are mapped.

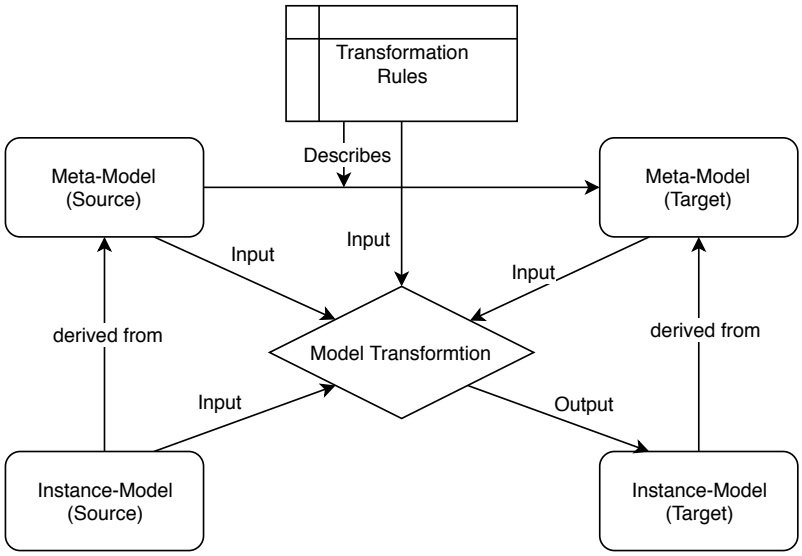


Figure 2.15: Outline of Model Transformation in MDD

Model-driven architecture (MDA) [58] is a set of guidelines for structuring software specifications in MDD. It can be seen as a subset of MDD based on the OMG group's standards.

In the software industry, many model transformation frameworks have been developed, such as Epsilon [50], VIATRA [8] and ATL [47]. These frameworks usually come with a rich set of tool-kits and support a wide range of functionalities. On the contrary, frameworks like Simple Transformer (SiTra) [2] provide lightweight solutions to model transformations. Despite the simplicity, SiTra provides everything needed for the work

described in this thesis. The use of SiTra in our work will be elaborated in Chapter 4.

2.6.1 SiTra: The Simple Transformer Library

Simple Transformer (SiTra) [2] is a minimal Java-based model transformation framework which consists of two interfaces, as shown in listing 2.1, and an implementation of a transformation algorithm. Usually in order to solve a transformation problem, multiple rules are needed. The implementation of the *check* method of a rule should return a value of *true* if the rule is applicable to the source object. The *build* method should construct a target object that the source object is mapped to. The *setProperties* method is called after the execution of the *build* method, which allows recursive calling of rules.

```
interface Rule<S,T> {
    boolean check(S source);
    T build(S source, Transformer t);
    void setProperties(T target, S source, Transformer t);
}
interface Transformer {
    Object transform(Object source);
    List<Object> transformAll(List<Object> sourceObjects);
    <S,T> T transform(Class<Rule<S,T>> ruleType, S source);
    <S,T> List<T> transformAll(Class<Rule<S,T>> ruleType,
}
```

Listing 2.1: Interfaces of SiTra

An example of model transformation using SiTra is the *Book and Paper to Publication Transformation*, as shown in Figure 2.16. In this example, *Books* and *Papers* are mapped to *Publications*. The sum of *numPages* (number of pages) of all *Chapters* in a book is mapped to the attribute *numPages* in a *Publication*. The implementation of the *Book* to *Publication* rule could be written as in listing 2.2. All implemented rules are then added into the transformer and a target *Publication* model can be generated automatically.

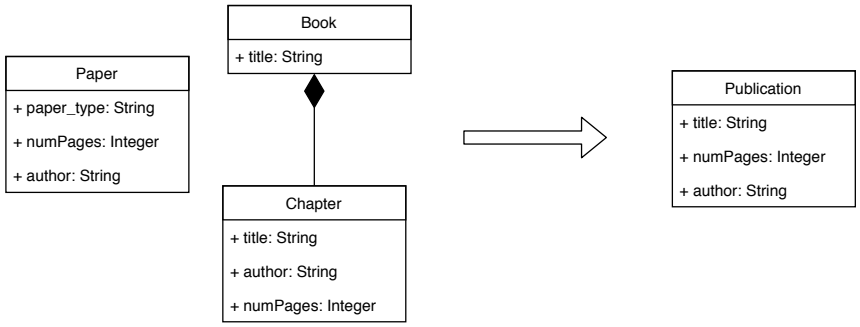


Figure 2.16: Model Transformation Example

```
class Book2Publication implements Rule<Book,Publication> {
    ...
    public Publication build(Book book, Transformer t) {
        Publication publ = new Publication( book.getTitle() );
        return publ;
    }
    public void setProperties(Publication publ,Book book,Transformer t){
        for(Chapter chapter: book.getChapters()){
            publ.numPages = publ.numPages + chapter.numPages;
        }
    }
}
```

Listing 2.2: Transformation Rule of SiTra

2.7 Chapter Summary

This chapter provides preliminary background for rule-based systems and formalises the RETE algorithm which is the fundamental algorithm behind many rule engines. It also provides an overview of the current rule-based event stream processing approaches, as well as their advantages and disadvantages.

In subsequent chapters, this thesis presents a technique to distribute the RETE algorithm in order to process larger event streams with rule-based systems .

CHAPTER 3

DISTRIBUTED EVENT PROCESSING WITH RULE-BASED SYSTEMS

In this chapter, we present a scalable and highly parallelised rule engine known as DRESS (Distributed Rule Engine on Spark Streaming). DRESS is built on top of the Apache Spark Streaming framework with an enhanced RETE algorithm for the processing of event streams. The main goal of DRESS is to improve the load-balancing and parallelism of RETE-based rule engines.

Section 3.1 describes the architecture of DRESS and introduces the techniques used in order to achieve better load-balancing and parallelism, such as micro-batching and dynamic job assignment.

In Section 3.2, we discuss the components of the DRESS applications and their interactions.

Section 3.3 elaborates on the construction of the DRESS networks and discusses the different types of DClusters that form an DRESS network.

This chapter is summarised in Section 3.4.

3.1 An Overview of the DRESS Architecture

In this section we discuss the architecture of DRESS. DRESS is built on top of the Apache Spark Streaming framework for the processing of event streams and uses the Apache

Kafka Framework [53] as the messaging system. Events are generated from multiple sources and inserted into the Kafka message queue. Thereafter, with the help of Spark Streaming, Kafka produces a stream of event batches. These batches are processed by the DRESS engine.

The computations of the batches are dynamically distributed to a cluster of executors managed by YARN. For failure recovery, the batches are check-pointed in memory or in the Hadoop Filesystem. Figure 3.1 shows the frameworks used by DRESS.

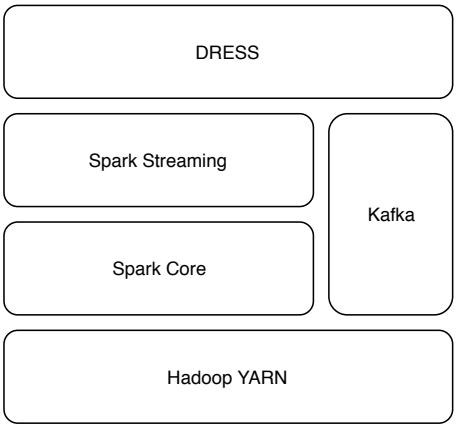


Figure 3.1: Tech Stack of DRESS

The architecture of DRESS is represented in Figure 3.2. A cluster of DRESS worker nodes consisting of one or more executors is managed by the YARN framework. A DRESS network (similar to a RETE network) is compiled from a set of rules at the client before it is submitted to the YARN Resource Manager (RM). Then, a DRESS application is created by the RM and assigned to a worker node which works as the DRESS master node. The DRESS master node consists of the DRESS application and the YARN Driver which manages the computing resources that are available for processing the batches. The DRESS application contains the DRESS network and a job scheduler which, through the YARN driver, distributes the jobs generated from the execution of the DRESS network to the executors. Finally, the outputs of the jobs are gathered by the DRESS master and are sent back to the client.

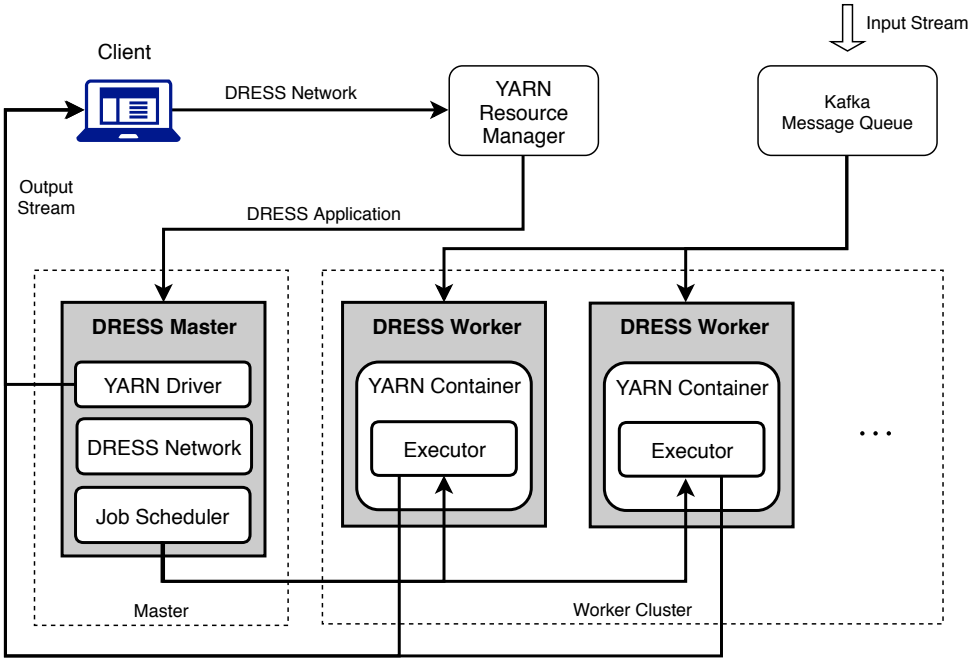


Figure 3.2: DRESS Architecture

3.1.1 Micro-batching

In contrast to the continuous operator model adopted by current rule engines, the computing nodes of a DRESS application process micro batches of data. Micro batching is the procedure which divides the incoming stream of events into groups of small batches. It has the benefits of batch processing (e.g. high throughput and high-level fault-tolerance) while, at the same time, keeping the latency of processing each event minimal. As shown in Figure 3.3, the incoming event stream is converted into a stream of small batches of events, and these batches are passed to and processed by the DRESS engine, which generates an output stream of batches.

In DRESS, the micro batches are created based on time intervals instead of size, in order to achieve a consistent minimal latency. More specifically, events received within a time interval (usually in milliseconds) are put into a batch. This time interval is configurable, which means by lowering its value we can ensure that any event does not have to wait too long before it is passed to the engine for processing. In the extreme cases, in which the time interval is set to zero, DRESS will behave like a continuous operator

model.

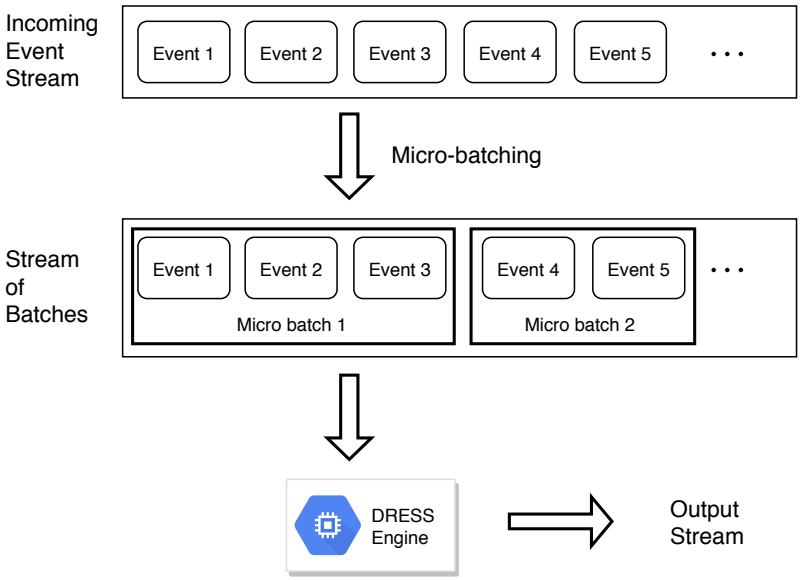


Figure 3.3: Micro-batching in DRESS

With Spark Streaming, these micro batches are represented by Resilient Distributed Datasets (RDDs). RDDs are immutable distributed collections of data. Due to its immutable nature, the processing of an RDD can be seen as the transformation from a source RDD to a target RDD. In Spark, RDDs are processed and stored across the worker cluster. As a result, the processing of an RDD may require an executor to copy the RDD from another executor, where it is stored, unless the RDD is in its local memory. This introduces an overhead of transferring data. However, this overhead can be minimised by dynamic job assignment with the optimisation of data locality, which is elaborated in the next section(s).

The streams of RDDs are represented by Discretised Streams (DStreams), which are maintained by the DRESS master. A DStream does not have the data of its RDDs. Instead, it holds their meta-information, including the sizes, locations, processing states and appointed executors. Once an RDD enters a DStream, the DRESS master creates a job and assigns it to an executor. Upon the completion of each job, the meta-information of the resulting RDD is sent back to the DRESS master.

3.1.2 Dynamic Job Assignment

Current approaches are based on the *static job assignment model*, in which the workload of a computing unit (such as an alpha node or a beta node) in the RETE networks is distributed to a static set of executors (workers). On the contrary, DRESS adopts a different dynamic job assignment paradigm [84, 18]. In DRESS, executors are no longer statically associated with any nodes of the RETE network. Instead, the executors are capable of completing all types of jobs originated from the RDDs (batches) of the DStreams. Furthermore, the job scheduler dynamically assigns the jobs to the executors according to their availabilities and workloads.

Figure 3.4 illustrates the dynamic job assignment process in DRESS. The structure of the DRESS network is maintained by the DRESS master. When an RDD arrives at any DStream of the DRESS network, the job scheduler creates a job to process it and assigns this job to available executors.

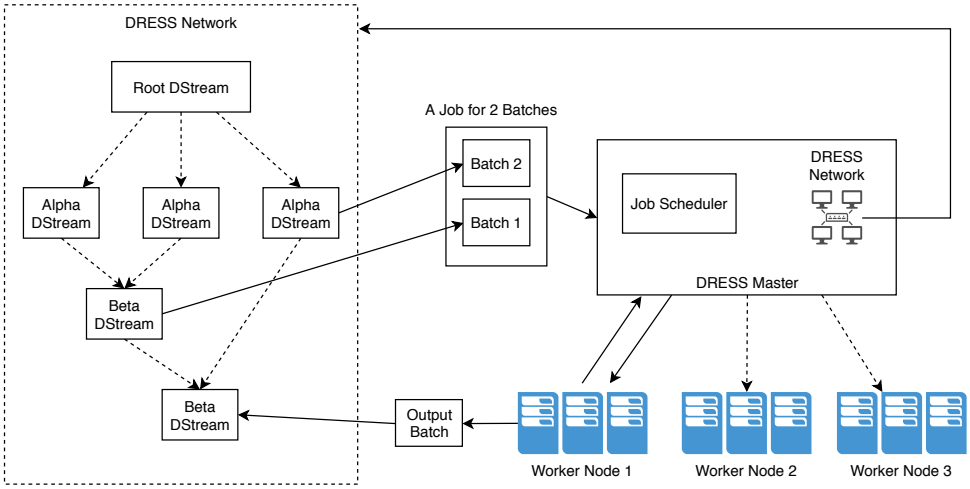


Figure 3.4: Dynamic Job Assignment for RETE Networks

The main advantage of the dynamic job assignment technique is that the workload is almost evenly spread out across the worker cluster, which removes the bottlenecks brought by a static job assignment strategy. However, this technique also comes with the overhead of transferring data to the executors. Especially, with the micro-batching model, this overhead becomes significant. DRESS minimises this overhead by enforcing

data locality.

Data locality refers to the concept of moving computation to the nodes where the data resides instead of moving data to computation. It is a strategy to increase the overall throughput and minimises the network congestion of distributed systems [39, 45, 46]. In DRESS, an executor is always preferred by the job scheduler to process the RDDs it has in its local memory. During the job scheduling process, it is likely that the executor which holds the RDD is occupied by other tasks. DRESS allows the job scheduler to wait a period of time to decide whether data locality can be achieved. If the executor becomes available while waiting, the RDD is scheduled to be processed on the executor. Otherwise, the RDD is transferred to another executor for processing.

The data locality of DRESS is realised with the help of YARN. The maximum time that the job scheduler can wait for enforcing data locality can be adjusted by different YARN configurations.

3.1.3 DRESS Worker Cluster (DCluster)

A DRESS executor cluster (DCluster) is the minimal computing unit of a DRESS application. It consists of a set of dynamically assigned executors from the Spark cluster, and the number of executors is adjusted according to the workload. A DCluster is responsible for completing a certain type of jobs that originated from one or more input DStreams. In Spark, the executors are generic to all types of jobs. As a result, an executor can be a part of more than one DCluster at the same time. The output of a DCluster is a DStream, which maintains the meta-information of RDDs containing the results of the jobs.

In Spark, the DClusters are created by defining a transformation from one or more (input) DStreams to another (output). More specifically, this transformation contains an operator that maps the RDDs of the input DStreams to RDDs of the output DStream. For DClusters with one input DStream, this can be done by the available operators provided by Spark, such as *map*, *filter* and *window*. For other DClusters, a customised operator needs to be defined. Consequently, the transformation can be seen as a function with two

parameters: the operator and a set of input DStreams.

During the execution of the DRESS network, the job scheduler creates jobs for the RDDs, which are received by DStreams, and assigns these jobs to available executors of the DCluster. A job consists of one or more RDDs from the input DStreams (one RDD for each DStream) and a procedure to process them. The executor processes these RDDs and produces an output RDD, which is stored in its local memory. Then, a notification with the location of the output RDD of the completed job is sent to the DRESS master. Finally, the meta-information of the output RDD is put into the output DStream of the DCluster and this output RDD will be used to create jobs for other DClusters.

3.2 DRESS Applications

A DRESS application is an instance of a rule-based expert system. As previously established, an expert system has three components - the rule base, the inference engine and the working memory. The rule base of a DRESS application can be constructed with current techniques such as declarative Domain Specific Languages (DSLs) or databases. The inference engine is an implementation of a DRESS network running on the Spark Streaming Framework. Moreover, the working memory is represented by RDDs and DStreams that are distributed across the cluster.

A DRESS application manages a DRESS network and a job scheduler. The DRESS network is a data structure of DStreams and DClusters that transform one DStream to another. When an RDD is inserted into a DStream in the network, the job scheduler makes arrangements for it to be processed remotely on an available executor. Then, the executor produces an output RDD and updates its meta-information to another DStream of the DRESS network.

3.3 DRESS Networks

DRESS can be seen as an enhanced RETE algorithm. As previously established, a DRESS application contains a DRESS network which is compiled from a set of rules. Typically, it consists of 4 types of DClusters:

- **Root DCluster**: the DCluster whose executors transform RDDs from the Kafka message queue to RDDs consisting of events in the unified format. The resulting RDDs are put into the root DStream.
- **Alpha DCluster**: the DClusters responsible for selecting events from an RDD based on simple conditional tests.
- **Beta DCluster**: the DClusters responsible for joining two DStreams and creating variable bindings among the events from the DStreams.
- **Terminal DCluster**: the special beta DClusters representing activations and performing actions of the rules.

Figure 3.5 shows the correspondences between the DRESS network and the RETE network. Each type of DClusters in the DRESS network corresponds to a type of nodes in the RETE network. Moreover, as DRESS adopts a micro-batching strategy to increase parallelism, the input and output of a DCluster are micro-batches (RDDs). The processing of these RDDs is distributed to available executors.

System	RETE				DRESS			
Model	RETE Network				DRESS Network			
Nodes	Name	Input	Output	Location of Processing	Name	Input	Output	Location of Processing
	Root Node	Event	Event	On the node	Root DCluster	RDD/ event	RDD/ event	On available executor
	Alpha Node	Event	Event	On the node	Alpha DCluster	RDD/ event	RDD/ event	On available executor
	Beta Node	Tuple	Tuple	On the node	Beta DCluster	RDD/ tuple	RDD/ tuple	On available executor
	Terminal Node	Tuple	Tuple	On the node	Terminal DCluster	RDD/ tuple	RDD/ tuple	On available executor

Figure 3.5: Correspondences between DRESS and RETE Networks

The DRESS network consists of 4 layers as shown in Figure 3.6. The first layer is the Kafka message queue which manages the input data from multiple sources. The second layer is the root DStream consisting of RDDs of events in a unified format. The third layer consists of alpha DStreams whose RDDs contain events that match or partially match a pattern. Finally, the last layer consists of beta DStreams whose RDDs represent logical compounds of patterns.

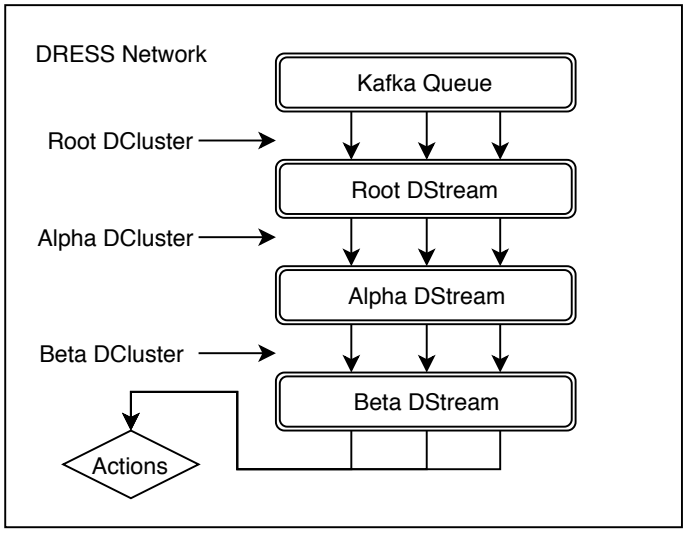


Figure 3.6: Layers of DRESS Networks

Figure 3.7 shows an example of a RETE network and a DRESS network compiled from the same rule with three patterns.

The following sections elaborate on the construction of the DRESS network from the rules and present each type of these DClusters.

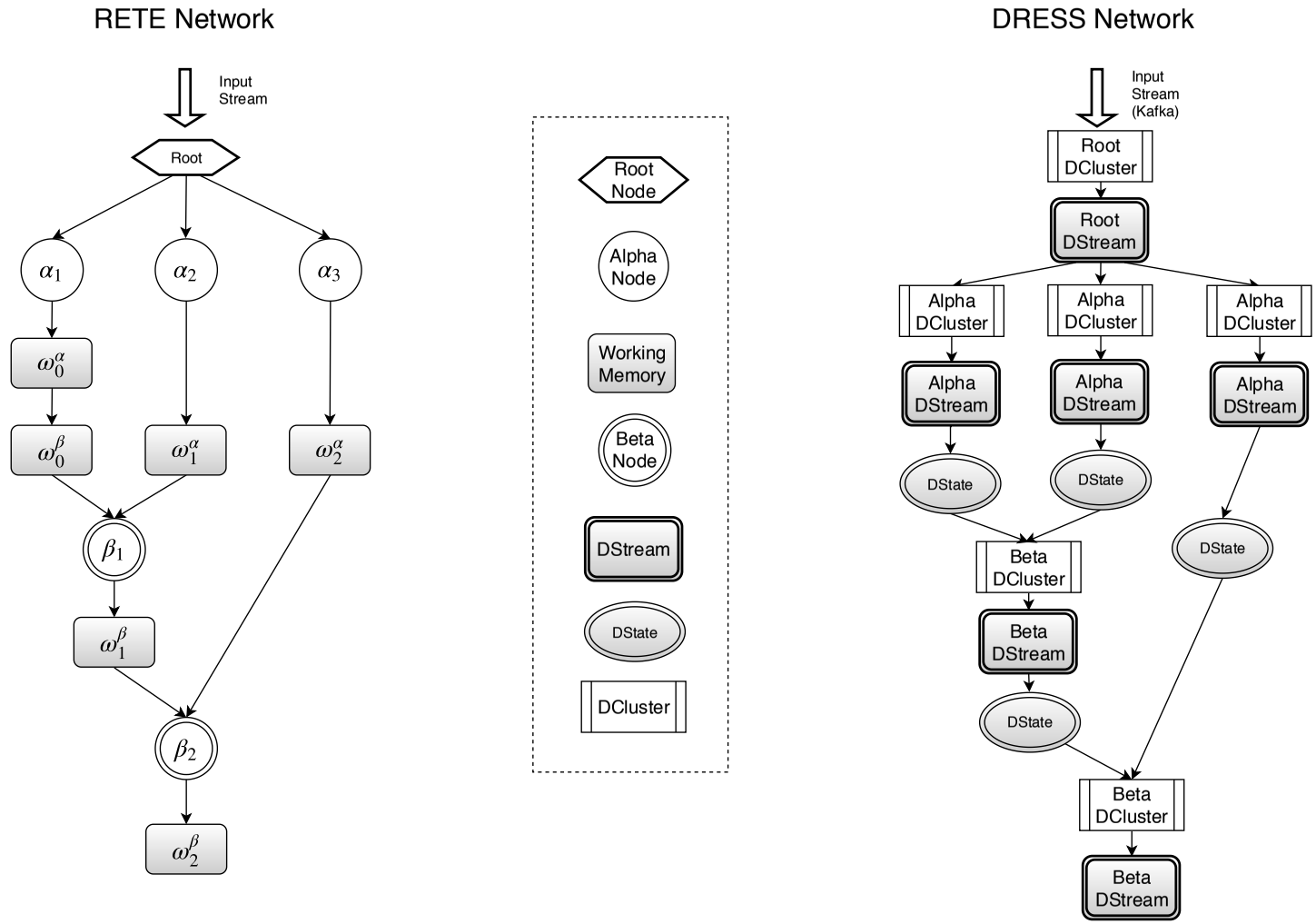


Figure 3.7: Example of RETE and DRESS Networks Compiled from the Same Rule

3.3.1 Root DCluster

Since the events from the Kafka message queue can be generated from multiple sources, their formats may be different. DRESS creates format adapters to convert them into a unified format. This unified format is a JSON-like dictionary which contains *type*, *index* and other data related fields of the events, as shown in listing 3.1.

```
{
    "type":"event_type",
    "index":Integer,
    "data_field_1":"value_1",
    "data_field_2":"value_2",
    ...
}
```

Listing 3.1: Unified Event Format

The root DCluster works as the entry point of a DRESS network. As previously established, a DCluster can be seen as a transformation from one or more input DStreams to an output DStream. The root DCluster transforms the event stream from the Kafka message queue to its output DStream, known as the root DStream. More specifically, the root DCluster applies the format adapter to each RDDs of the input DStream, and therefore the root DStream consists of batches of events in the unified format.

As shown in Figure 3.8, when an RDD is produced by the Kafka message queue, Kafka notifies the DRESS master. The DRESS master creates a job consisting of the format adapter and the RDD. Afterwards, the job is assigned to an executor appointed by the job scheduler. Then, the executor converts the events of different formats in that RDD to the unified format. Finally, the executor produces an output RDD whose meta-information is inserted into the root DStream.

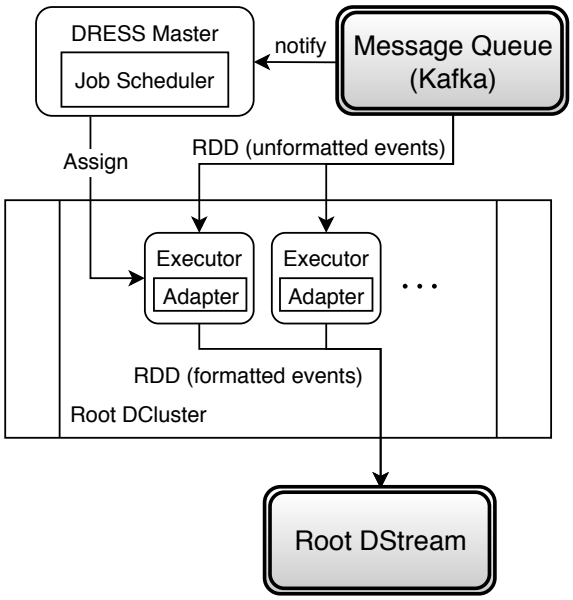


Figure 3.8: DRESS - Root DCluster

3.3.2 Alpha DCluster (1-input DCluster)

The alpha DClusters of the DRESS network correspond to the alpha nodes of the RETE network. They are responsible for testing conditional attributes of the events. An alpha DCluster is also called a 1-input DCluster as it generates an alpha DStream from processing RDDs of a single input DStream. Furthermore, the input of the alpha DClusters can either be the root DStream or an alpha DStream.

Consider a pattern P whose propositions p_i test different attributes of a given event. DRESS creates an alpha DCluster (as well as its output alpha DStream) for each proposition p_i and connects the created alpha DClusters into an *alpha chain*. Upon the arrival of an RDD at the root DStream, an executor of the first alpha DCluster in the chain is activated to select events from the RDD satisfying the corresponding proposition. These events are inserted into the output RDD, which is sent to the next alpha DCluster for the test of another attribute. Each alpha chain created from a pattern P terminates at a DState, which is the accumulation of RDDs from the alpha DStream of the last alpha DCluster.

Individually, the implementation of an alpha DCluster is similar to the implementation of the root DCluster (Figure 3.8), except that they have different input DStreams and the

executors are responsible for different types of jobs.

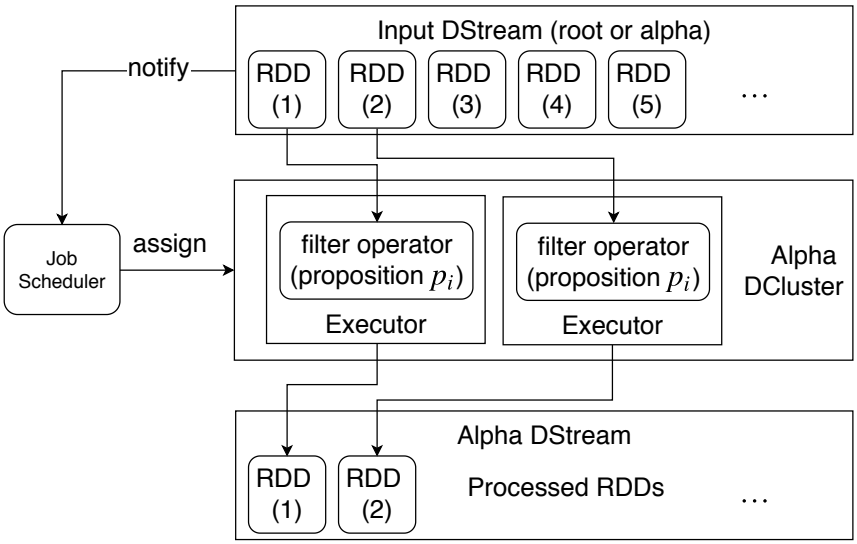


Figure 3.9: DRESS - Alpha DCluster

Consequently, an atomic rule $r = P_1 \wedge P_2 \wedge \cdots \wedge P_n \rightarrow h$ will result in a set of alpha chains. Each of them can be seen as a sequence of transformations from the root DStream to the alpha DStream whose RDDs consist of events matching a pattern.

To transform the DStreams, the DRESS master creates jobs using the *filter* operator provided by Spark, as shown in Figure 3.9. The *filter* operator creates one job for each RDD of the input DStream and the job is completed by an available executor of the alpha DCluster. Each job consists of the corresponding proposition of the alpha DCluster and an input RDD. The procedure to complete the job is shown in Algorithm 1.

Algorithm 1 Execute alpha jobs

```
1: function ALPHA_EXECUTOR(rdd, proposition)
2:     output ← emptyRDD
3:     for each element in rdd do
4:         if proposition(element) then
5:             append element to output
6:         end if
7:     end for
8:     return output
9: end function
```

3.3.3 Beta DCluster (2-input DCluster)

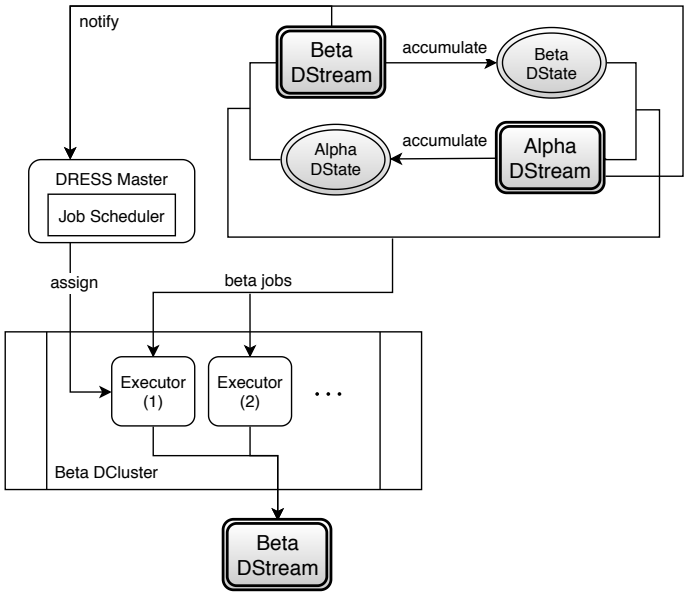


Figure 3.10: DRESS - Beta DCluster

The beta DClusters (also called 2-input DClusters) of the DRESS network correspond to the beta nodes of the RETE networks. A beta DCluster has a left input and a right input, where the left is a beta DStream and the right is an alpha DStream. As DStreams are stateless (i.e. they only keep RDDs that are not processed by the system), both input DStreams are accumulated to corresponding DStates, which contain all the RDDs they have received, as shown in Figure 3.10.

A customised operator *join* is defined by the DRESS master for job creation. When an RDD arrives at one of the two input DStreams of a beta DCluster, the DRESS master is notified. This RDD is paired with every RDD in the DState corresponding to the other input DStream. Then, the operator creates one job for each of these RDD pairs.

A beta job consists of two input RDDs (one from each input DStream) and the set of variable bindings in the system, as shown in Figure 3.11. Similar to the output of the beta nodes in RETE networks, the RDDs of the beta DStreams contain tuples of events matching a compound of patterns, while the RDDs of the alpha DStreams contain events matching a single pattern. Consider an atomic rule $r = P_1 \wedge_1 P_2 \wedge_2 \cdots \wedge_{n-1} P_n \rightarrow h$. Each logical conjunction $\wedge_i$ joins the beta DStream containing RDDs of tuples of events

matching the compound $P_1 \wedge_1 \cdots \wedge_{i-1} P_i$, and the alpha DStream containing RDDs of events matching the pattern P_{i+1}.

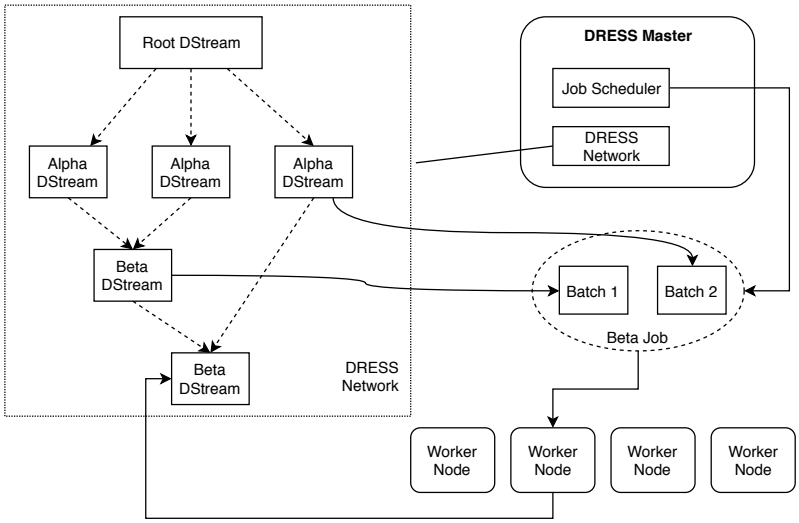


Figure 3.11: Beta Job in DRESS

Let rdd_1 be the batch of event tuples matching the compound $P_1 \wedge_1 \cdots \wedge_{i-1} P_i$ and rdd_2 be the batch of events matching P_{i+1}. If there is a variable binding *vb* that binds a variable in the pattern P_{i+1} to a variable in a pattern $P_j, 1 \leq j \leq i-1$, then the variable binding *vb* is applied to tuples of elements in rdd_1 and rdd_2. The procedure to complete a beta job is shown in algorithm 2.

The beta DClusters with variable bindings are called *variable binding* DClusters, while other beta DClusters are called *join* DClusters.

3.3.4 Terminal DCluster

The terminal DClusters are a special type of beta DClusters whose output DStreams are not processed further. They correspond to the terminal nodes of the RETE networks which represent the activation of rules and perform the actions.

For an atomic rule $r = P_1 \wedge_1 P_2 \wedge_2 \cdots \wedge_{n-1} P_n \rightarrow h$, one beta DCluster is created for each logical conjunction $\wedge_i$, joining the compound $P_1 \wedge_1 \cdots \wedge_{i-1} P_i$ to the pattern P_{i+1}. The beta DCluster created for the last conjunction $\wedge_{n-1}$ is called the terminal DCluster of

Algorithm 2 Execute beta jobs

```
1: function BETA_EXECUTOR(rdd1, rdd2, variable_bindings)
2:     output ← emptyRDD
3:     for each x in rdd1 do
4:         for each y in rdd2 do
5:             for each vb in variable_bindings do
6:                 pattern_1 ← left(vb)   ▷ first pattern in the variable binding statement
7:                 pattern_2 ← right(vb)          ▷ second pattern in the variable binding
8:                 if x matches pattern_1 then
9:                     if y matches pattern_2 then
10:                        if vb(x, y) then
11:                            t ← tuple(x, y)
12:                            append t to output
13:                        end if
14:                    end if
15:                end if
16:            end for
17:        end for
18:    end for
19:    return output
20: end function
```

the rule r.

An RDD of the output DStream of a terminal DCluster consists of tuples of events matching the conditions of the rule r. Hence, the action h needs to be performed for every tuple in the RDD. A job is created by the DRESS master for each arrival of an RDD to the output DStream of the terminal DClusters. The procedure to complete the job is shown in algorithm 3.

Algorithm 3 Perform actions

```
1: function TERMINAL_EXECUTOR(rdd, h)
2:     for each element in rdd do
3:         h(element)                          ▷ perform the action h for the tuples
4:     end for
5: end function
```

3.4 Chapter Summary

This chapter presents the proposed rule-based system DRESS for large event stream processing. It begins with the architecture of DRESS and strategies to improve the performance of a rule engine, namely dynamic job assignment and micro-batching. It also presents the the DCluster which is the minimal computing unit of a DRESS network. This is followed by the introduction to DRESS applications and their executions over the DRESS architecture. Then, this chapter elaborates the algorithm to construct DRESS networks from the rules.

Event stream processing with DRESS can be summarised as following steps:

1. A set of rules is designed by human experts. The rules are converted to atomic rules using logical equivalence laws. Each atomic rule has several variable bindings and patterns connected by logical conjunctions. An atomic rule is compiled to a DRESS network.

2. Each pattern of an atomic rule is compiled to an alpha chain of alpha DClusters, each of which tests one attribute of a given event. An alpha chain selects events that match its corresponding pattern.

3. The beta DClusters join the events matching different patterns and implement variable bindings.

4. Terminal DClusters are special beta DClusters, whose output is not processed further. Instead, Terminal DClusters represent the activations of rules and perform actions.

CHAPTER 4

AUTOMATED TRANSFORMATION FROM RETE TO DRESS

The transformation from RETE to DRESS networks is time-consuming and prone to human error. Therefore, this work utilises an MDA based approach to automate this transformation with the help of the SiTra framework.

Section 4.1 introduces the MDA technique. In section 4.2 and 4.3, the meta models for RETE and DRESS networks are presented. The transformation rules, which map the elements of the RETE network into their corresponding elements in the DRESS network, are described in section 4.4.

The SiTra implementation of the transformer is illustrated in section 4.5.

4.1 MDA-based Transformation for RETE Networks

In MDA, a meta-model can be understood as an abstraction of a class of models. It describes the types of model elements and their interactions. For example, the root node of the RETE network meta-model in Figure 4.2 interacts with the alpha node, while the alpha node interacts with the root node, the alpha node itself and the alpha memory. Models conforming to the meta-model are concrete instances of the meta-model. For instance, all RETE networks presented in this thesis are concrete instances of the RETE network meta-model.

Given the meta-models of the source and the target models, and a set of transformation rules [1] that map the model elements and their interactions from the source meta-model to the target meta-model, the transformation can be automated. Figure 4.1 shows the transformation process to the DRESS networks from their RETE counterparts generated by Drools. The RETE and DRESS networks conform to their corresponding meta-models respectively. A model transformer directs the transformation of a given concrete RETE network to the target DRESS network, according to the mapping on the meta-model level.

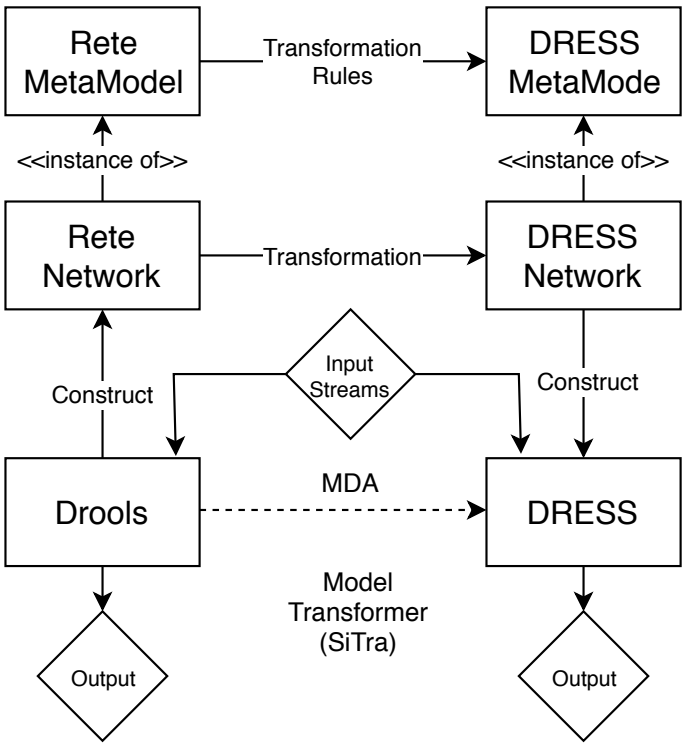


Figure 4.1: Automated Transformation from RETE to DRESS

4.2 Meta-model for RETE Networks

The meta-model for RETE networks is presented in Figure 4.2. A `RootNode` interacts with zero or more `AlphaNodes`. An `AlphaNode` usually interacts with one

[1] In this thesis, the word 'rule' is used in the different contexts of rule-based systems and model transformations. As they have different meanings, we use the phrase 'transformation rule' when there is a chance of confusion.

RootNode and one AlphaMemory, reflecting the scenario where the root node broadcasts events streams to the alpha nodes and the alpha nodes pass their results to alpha memories. It is, however, possible for an AlphaNode to have no interaction with RootNodes and AlphaMemories. This is due to the fact that a chain of alpha nodes can be created for checking different attributes of the same event and the alpha nodes in the middle of the chain may interact with other alpha nodes only (see section 2.1.5, page 22).

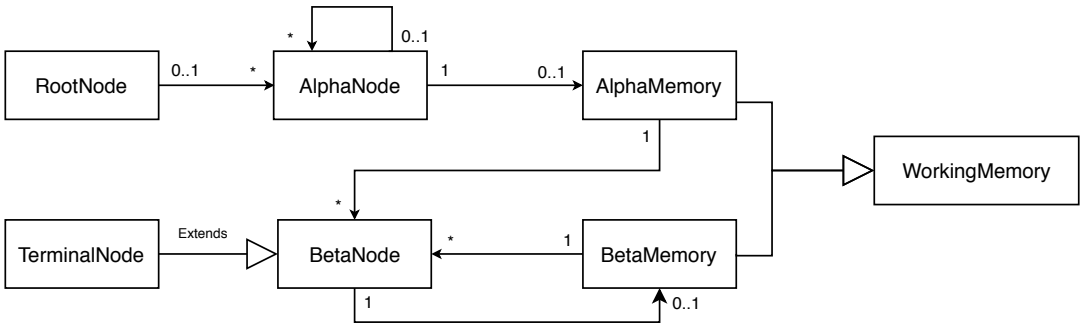


Figure 4.2: The RETE Network Meta-model

A WorkingMemory is an abstract class, which is further extended by an AlphaMemory and a BetaMemory. A BetaNode accepts an AlphaMemory and a BetaMemory as input, and forwards its results to a BetaMemory.

A TerminalNode is a specialised BetaNode whose results are not stored in a BetaMemory and are sent to the output stream of the system.

An instance of the RETE network meta-model represents a concrete RETE network. The abstract syntax of the RETE network in Figure 3.7 (page 51) can be captured as an instance (figure 4.3) of the meta-model. This abstract syntax shows that the RootNode interacts with three AlphaNodes, and the AlphaNodes α_1, α_2 and α_3 send their results to AlphaMemories ω_0^α, ω_1^α and ω_2^α respectively. Moreover, each of the BetaNodes β_1 and β_2 accepts one AlphaMemory and one BetaMemory as input, and generates one BetaMemory.

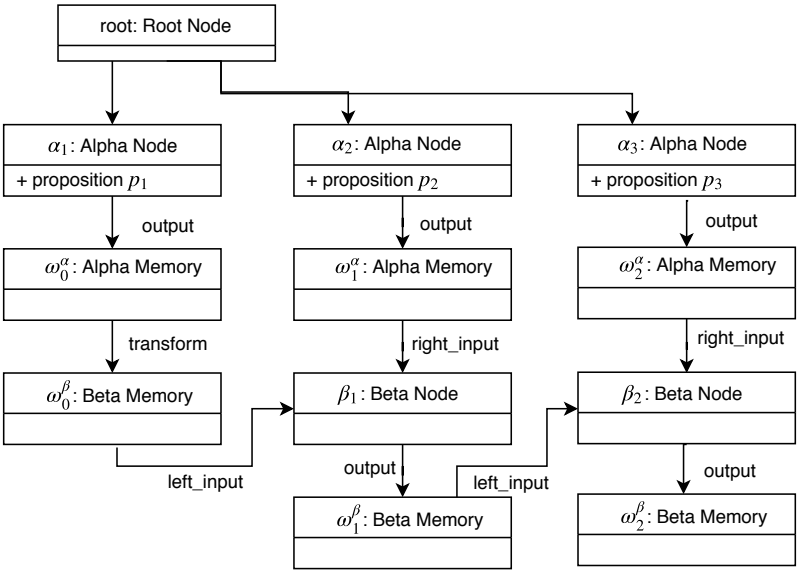


Figure 4.3: Abstract Syntax of RETE Networks

4.3 Meta-model for DRESS Networks

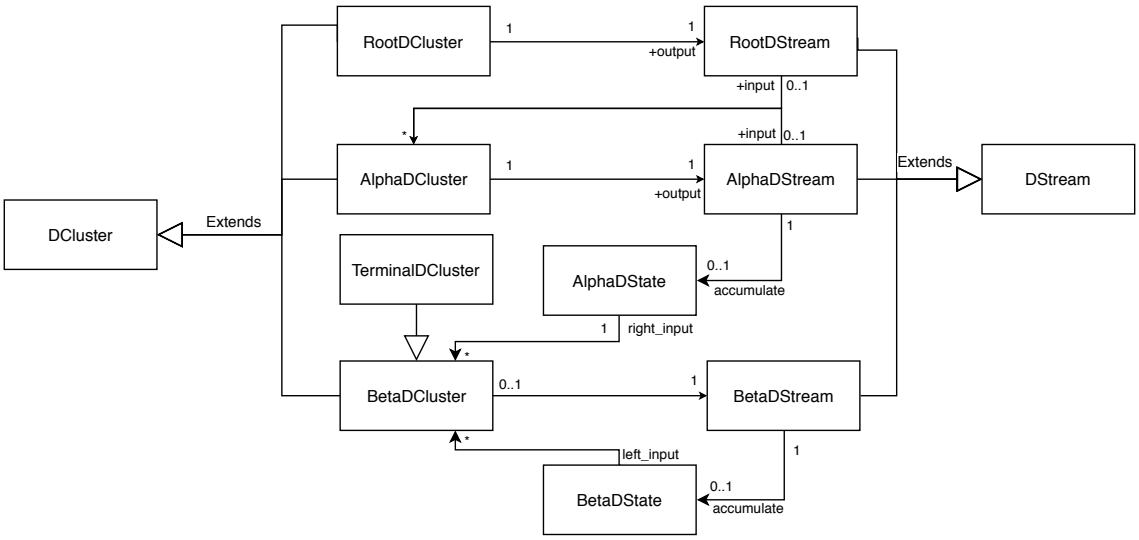


Figure 4.4: The DRESS Network Meta-model

Figure 4.4 shows the meta-model for the DRESS networks. A `DCluster` is an abstract class, which is specialised into a `RootDCluster`, an `AlphaDCluster` and a `BetaDCluster`. Moreover, a `DStream` is also an abstract class which is specialised into a `RootDStream`, an `AlphaDStream` and a `BetaDStream`. DClusters of each type send the outputs to their corresponding DStreams. An `AlphaDCluster` accepts either a `RootDStream` or an `AlphaDStream` as its input and a `BetaDCluster` has

a `BetaDStream` as its left input, and an `AlphaDStream` as its right input. Finally, a `TerminalDCluster` is a specialised `BetaDCluster` which represents the activations of rules of the DRESS network.

4.4 Transformation Rules

This section describes the model transformation process, whereby any RETE networks conforming to the RETE network meta-model in Figure 4.2 are transformed into DRESS networks. This requires a set of four transformation rules mapping the elements of the RETE networks into DRESS networks. Figure 4.5 presents an overview of the correspondence between the elements in RETE and DRESS networks.

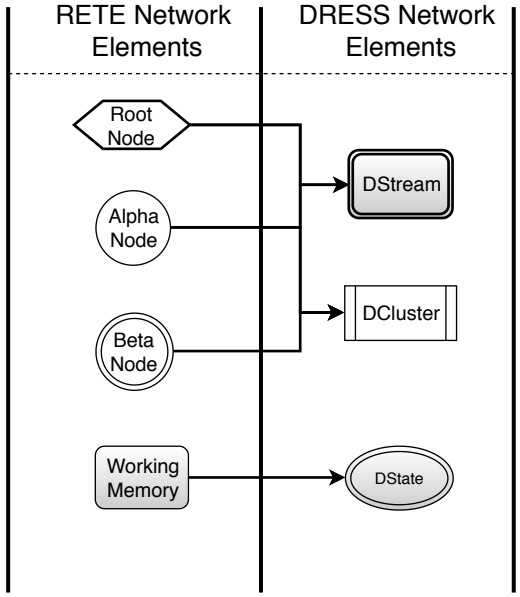


Figure 4.5: Correspondence between RETE and DRESS Networks

4.4.1 Rule 1: Transforming Root Nodes

The transformation maps a `RootNode` of the RETE meta-model into a `RootDCluster` and a `RootDStream` in the DRESS meta-model, as shown in Figure 4.6. The `RootDCluster` of the DRESS network at the model level consists of Spark executors that are responsible for converting the events from the `kafkaStream` into a unified format. The `map` func-

tion creates jobs for the batches of events and assigns the `adapter` function to an available Spark executors. The output of formatted batches are passed to the `rootDStream`, as shown in the target Spark code.

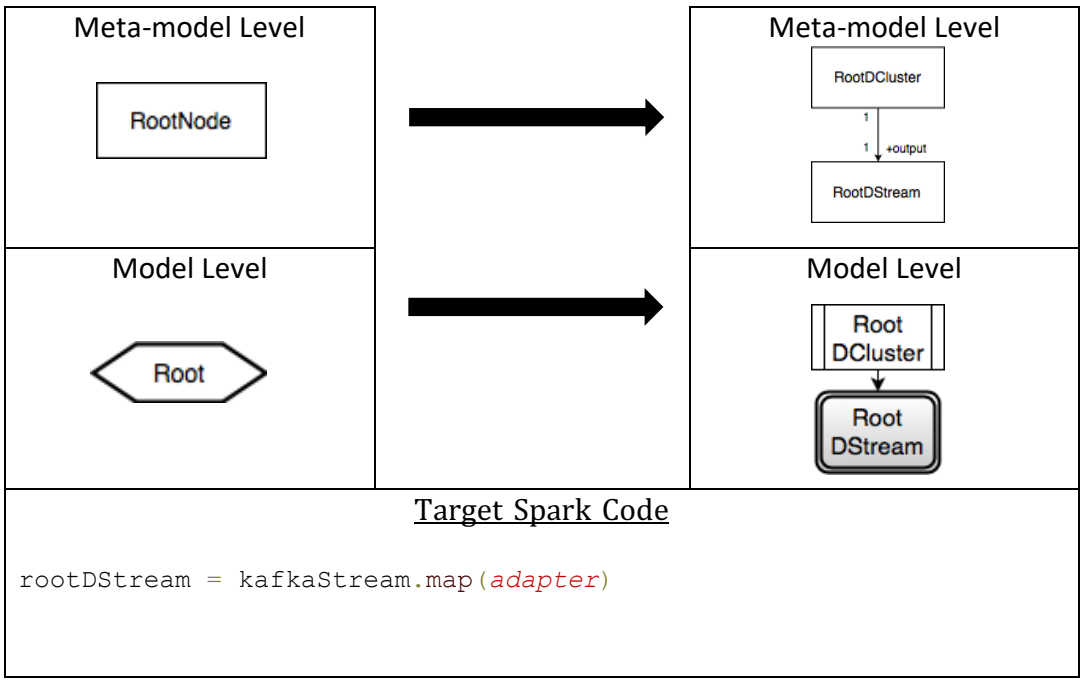


Figure 4.6: Transformation Rule - Root Nodes

4.4.2 Rule 2: Transforming Alpha Nodes

The transformation rule for alpha nodes in this approach involves the transformations of the input and the output of the alpha nodes, as well as the alpha nodes themselves. In DRESS networks, an `AlphaDCluster` has precisely one input DStream and one output `AlphaDStream`. Note that the input of an `AlphaDCluster` can be either the `RootDStream` or an `AlphaDStream` generated by another `AlphaDCluster`, as shown in Figure 4.7. If an `AlphaDCluster` is originated (transformed) from an alpha node, which is the last node of an alpha chain, in the RETE network, the `AlphaDStream` generated by the `AlphaDCluster` is accumulated into an `AlphaDState`.

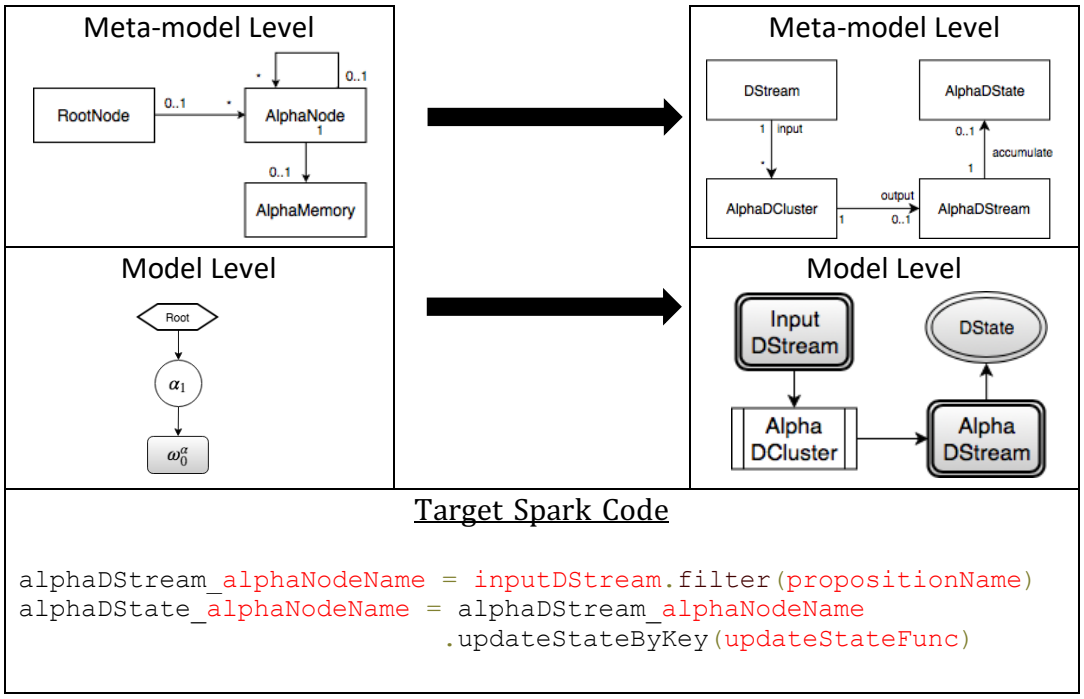


Figure 4.7: Transformation Rule - Alpha Nodes

Remark. *In Spark, a DStream can be accumulated into a DState using the updateStateByKey method. This method requires an function describing how new updates to previous states are processed.*

The input and the output of an alpha node are mapped to the input and output DStream of the `AlphaDCluster`. The alpha node itself is mapped to an `AlphaDCluster` consists of a filter operator. The proposition corresponding to the alpha node is used to create the filter operator, as shown in the target Spark code section of Figure 4.7. For example, the alpha nodes of the RETE network in figure 3.7 will be transformed into the following Spark code:

```
alphaDStream_a1 = rootDStream.filter(proposition_a1)
alphaDStream_a2 = rootDStream.filter(proposition_a2)
alphaDStream_a3 = rootDStream.filter(proposition_a3)

alphaDState_a1 = alphaDStream_a1.updateStateByKey(updateStateFunc)
alphaDState_a2 = alphaDStream_a2.updateStateByKey(updateStateFunc)
alphaDState_a3 = alphaDStream_a3.updateStateByKey(updateStateFunc)
```

4.4.3 Rule 3: Transforming Beta Nodes

The transformation for beta nodes involves two types of model elements - the beta memories and the beta nodes. The input alpha memories and beta memories are transformed into `AlphaDStates` and `BetaDStates` respectively. In addition, the beta nodes are transformed into `BetaDClusters`. Moreover, the output beta memory of a beta node is transformed into a `BetaDStream`, which is then accumulated into a `BetaDState`.

As shown in Figure 4.8, a `BetaDCluster` is created to aggregate two `BetaDStates`. This is reflected in the *target spark code* section of the figure, where the `leftInputDState` is joined with the `rightInputDState`. A join function *joinFunc* is created to direct the aggregation of the two inputs.

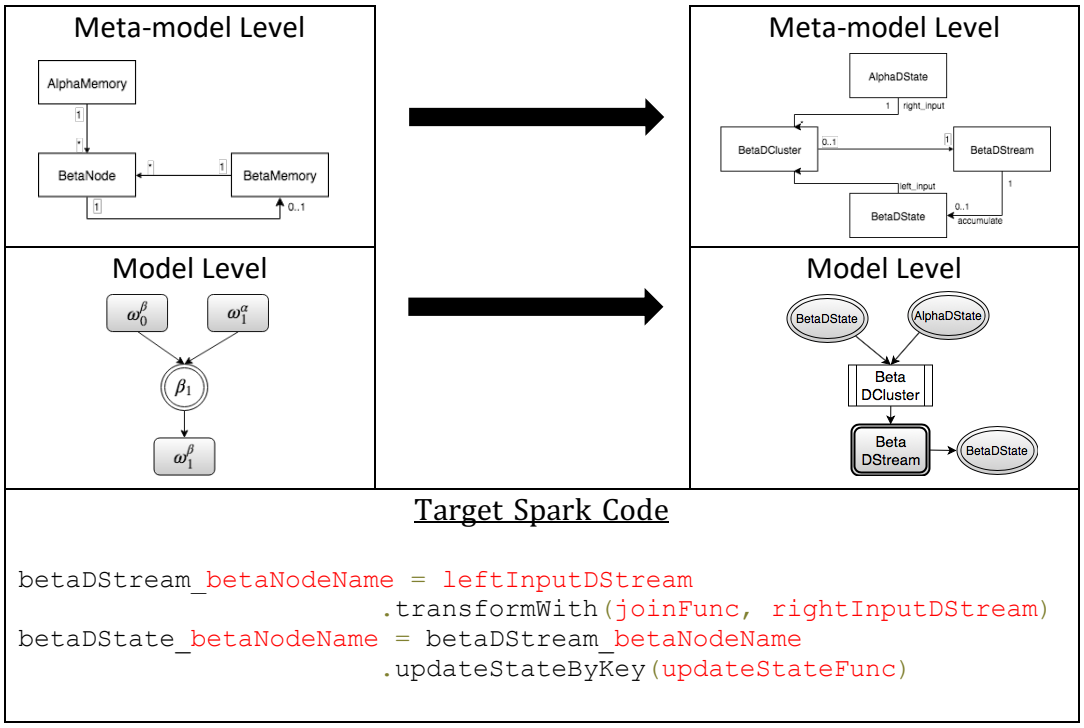


Figure 4.8: Transformation Rule - Beta Nodes

4.4.4 Rule 4: Transforming Terminal Nodes

`TerminalDClusters` are a special type of `BetaDClusters`. Therefore, they are transformed using Rule 3. As previously established, each `BetaDCluster` has an output DStream. RDDs in the output DStream of the `TerminalDClusters` consists of tuples that satisfy the conditions and represents the activation of a rule. With Spark Streaming, this activation is done by applying the action function h on every tuple in these RDDS.

An example of the target spark code in which the output of the `TerminalDCluster` is simply printed out is shown as follows:

```
def action1(rdd):
    rdd.foreach(print)

betaDStream_terminal1.foreachRDD(action1)
```

In this example, the function *action*1 is applied to every RDD (batch) in the `BetaDStream`. And the function *action*1 prints out the content of every record in the batch. In real life scenarios, this action function is taken from the RETE network which is compiled from the rules. Moreover, this function can perform different types of actions, e.g. inserting new facts (events) to the system, or passing the records to other systems for future use.

4.5 Transforming RETE to DRESS with SiTra

As previously established, the transformation rules map each element of the RETE meta-model to an element of the DRESS meta-model. The automation of the transformation can be made by implementing these rules in SiTra and using its transformer. This section elaborates the implementation of the SiTra rule that transforms the root node of RETE networks to root DClusters of DRESS network. The complete SiTra rules can be found in appendix A.

Root Node to RootDCluster

The RootNode of a RETE network is transformed into a RootDCluster with its output RootDStream in the target DRESS network. This can be achieved by implementing a SiTra rule in which the *build* method creates the target RootDCluster and RootDStream, as shown in listing 4.1. The relation between the created RootDCluster and RootDStream is also defined in this SiTra rule by adding the RootDStream to the RootDCluster's children list.

```
public class RuleRootNode implements Rule<RootNode, RootDCluster>{
    ...
    public RootDCluster build(RootNode source, Transformer t) {
        RootDCluster rootDCluster = new RootDCluster("root");
        DStream dStream_root = new DStream("root");
        rootDCluster.addChild(dStream_root);
        return rootDCluster;
    }

    public void setProperties(RootDCluster target, RootNode source, ...) {
        for(Object node: t.transformAll(source.getChildren())){
            AlphaDCluster alphaDCluster = (AlphaDCluster) node;
            DStream rootDStream = target.getChildren().get(0);
            rootDStream.addChild(alphaDCluster);
        }
    }
}
```

Listing 4.1: SiTra Rule: RootNode to RootDCluster

According to the RETE meta-model, a RootNode interacts with zero or more AlphaNodes. This interaction is transformed in the *setProperties* function after the transformation of the RootNode is finished. Given the source RootNode and the target RootDCluster, the transformer recursively transforms the AlphaNodes which interact with the RootNode. Then the interactions between these AlphaNodes and the RootNode are transformed into the interactions between the AlphaDClusters and the output DStream of the RootDCluster.

4.6 Chapter Summary

This chapter introduces an automated transformation from current RETE based models to DRESS models. This transformation is based on MDA and meta-modelling. In section 4.2 and section 4.3, the meta-models for RETE and DRESS networks are presented. Section 4.4 describes the transformation rules that map each element of the RETE meta-model to an element of the DRESS meta-model.

Section 4.5 obtains the target DRESS networks by applying the transformation rules to RETE networks with the SiTra library.

CHAPTER 5

VERIFICATION OF DISTRIBUTED RULE ENGINES

As an attempt to distribute a RETE-based system, the implementation of DRESS needs to ensure that, given the same input, it outputs exactly the same stream as the original RETE system. This chapter describes the method that can be used to verify the correctness of DRESS.

The verification method has two parts. First, in section 5.2, we verify the correctness of DRESS without considering the order of the output. Second, in section 5.3, we verify the preservation of the output order in the proposed model DRESS.

5.1 Formalising DRESS Networks

Both the original RETE system described in Forgy's paper [31] and DRESS are special cases of the general RETE-based Systems described in section 2.1. In this section, we formalise DRESS networks.

5.1.1 Alpha DCluster

In DRESS, each proposition p_i of a pattern P is represented by an alpha DCluster $C^{\alpha}_{p_i} = \{\alpha^1_{p_i}, \ldots, \alpha^n_{p_i}\}$ of executors, where n is the cluster size. Mathematically, an executor $\alpha^j_{p_i}$ corresponding to the proposition p_i receives a sub-stream of the input stream $S\restriction_A : A \to \mathbb{E}$

of the alpha DCluster $C^{\alpha}_{p_i}$, producing an alpha stream $S\restriction_{A^j_{p_1}}$.

An executor of the alpha DCluster works in the same way as a RETE alpha node. All executors in the same alpha DCluster are assigned to the same proposition p_i. Consider an n-partition $\{X'_1,\ldots,X'_n\}$ of the the domain of $S|_A$. The output of the alpha DCluster $C^{\alpha}_{p_i}$ corresponding to proposition p_i can be seen as a set of alpha functions:

$$C^{\alpha}_{p_i} = \left\{ \begin{array}{l} S|_{A^1_{p_i}} : A^1_{p_i} \to \mathbb{E}, \text{where } A^1_{p_i} \subseteq X'_1 \wedge \forall e \in img(S|_{A^1_{p_i}}) : p_i(e) \\ \ldots \\ S|_{A^n_{p_i}} : A^n_{p_i} \to \mathbb{E}, \text{where } A^n_{p_i} \subseteq X'_n \wedge \forall e \in img(S|_{A^n_{p_i}}) : p_i(e) \end{array} \right\}$$

The pattern P is represented by an alpha chain of alpha DClusters. Every RDD of the root DStream $S^{in} : \mathbb{N} \to \mathbb{E}$ flows through the chain and is processed by one executor of each alpha DCluster. Hence, given an n-partition $\{X_1,\ldots,X_n\}$ of $\mathbb{N}$, the output of an alpha chain corresponding to pattern P can be modelled by the following set of alpha functions:

$$C^{\alpha}_{P} = \left\{ \begin{array}{l} S|_{A^1_P} : A^1_P \to \mathbb{E}, \text{where } A^1_P \subseteq X_1 \wedge \forall e \in img(S|_{A^1_P}) : \bigwedge\limits_{p_i \in P} p_i(e) \\ \ldots \\ S|_{A^n_P} : A^n_P \to \mathbb{E}, \text{where } A^n_P \subseteq X_n \wedge \forall e \in img(S|_{A^n_P}) : \bigwedge\limits_{p_i \in P} p_i(e) \end{array} \right\}$$

The output of an alpha chain is accumulated to a DState. Let $\uplus^{\alpha}$ be the operator that accumulates the output of alpha DClusters. The above set of alpha functions is aggregated into a single alpha function $\uplus^{\alpha}(C^{\alpha}_P) = S|_{A_P} : A_P \to \mathbb{E}$ satisfying: $A_P = \bigcup_{j=1}^{n} A^j_P$, and $S|_{A_P}(k) = e_k \iff \exists S|_{A^j_P} \in C^{\alpha}_P : S|_{A^j_P}(k) = e_k$.

5.1.2 Beta DCluster

In DRESS, a beta DCluster joins a beta DStream and an alpha DStream. Consider the scenarios in which an RDD arrives at the left input (beta DStream) of a beta DCluster.

Each executor in the beta DCluster receives a sub-stream of the input beta DStream, and receives the accumulation (DState) of the input alpha DStream, as indicated in Figure 5.1. Similarly, in the scenarios where an RDD arrives at the right input (alpha DStream) of the beta DCluster, the executor receives a sub-stream of the input alpha DStream, and receives the accumulation of the input beta DStream.

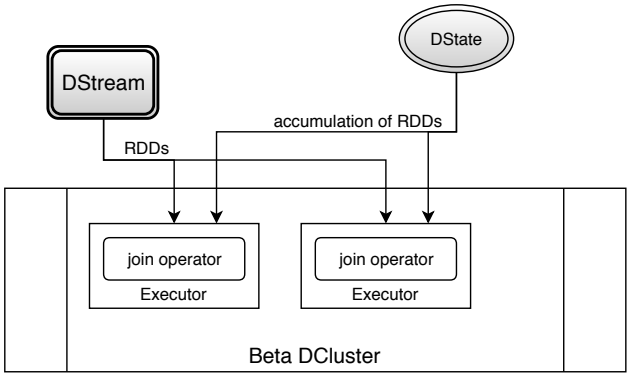


Figure 5.1: Distributed Beta DCluster

A beta DCluster $C_i^{\beta} = (\beta_i^1, \ldots, \beta_i^n)$ consists of n executors corresponding to the beta node β_i of the original RETE network. Then, the output of a beta DCluster C_i^{β} can be modelled by a set of beta functions $S_j^{\beta_i} : \mathbb{N} \rightarrow \mathbb{T}$ mapping an index to a tuple.

Given a beta DCluster C_{i-1}^{β} and an alpha DCluster C_{i+1}^{α}, each executor in the beta DCluster C_i^{β} joins the output of a function $S_j^{\beta_{i-1}}$ in C_{i-1}^{β} and the output of $\uplus^{\alpha}(C_{i+1}^{\alpha})$, yielding

$$C_i^{\beta} = \left\{ \begin{array}{l} S_1^{\beta_i} : \mathbb{N} \rightarrow img(S_1^{\beta_{i-1}}) \times img(\uplus^{\alpha}(C_{i+1}^{\alpha})) \\ \ldots \\ S_n^{\beta_i} : \mathbb{N} \rightarrow img(S_n^{\beta_{i-1}}) \times img(\uplus^{\alpha}(C_{i+1}^{\alpha})) \end{array} \right\}$$

Let $\uplus^{\beta}$ be the operator that accumulates the output of beta DClusters and $\sigma(t)$ be the largest index among all events of a tuple t. The above set of beta functions is aggregated

into a single beta function $\uplus^{\beta}(C_i^{\beta}) = S^{\beta_i} : \mathbb{N} \to \mathbb{T}$ satisfying:

$$t_x = S^{\beta_i}(x) \iff \exists j \exists S_k^{\beta_i} \in C_i^{\beta} : t_x = S_k^{\beta_i}(j), \text{ and,} \tag{5.1}$$

$$(t_x = S^{\beta_i}(x)) \wedge (t_y = S^{\beta_i}(y)) \wedge (x < y) \iff \sigma(t_x) < \sigma(t_y). \tag{5.2}$$

5.2 Orderless Equivalence Between RETE and DRESS

For certain applications of rule-based systems, the order of output is not required. For example, in a hospital system which produces a daily list of patients who have visited the hospital, the ordering of the patient list is not required because after all what we want is a list of names.

In this section, we verify the correctness of the transformation from RETE to DRESS without considering the order of their output. More specifically, we verify that, given the same input, DRESS generates the same set of rule activations as RETE, even though the activations themselves may not come in the same order. We call this *orderless equivalence*.

Definition 5.1. *Given two RETE-based systems R and R', as well as their output streams represented by functions $S : \mathbb{N} \to \mathbb{T}$ and $S' : \mathbb{N} \to \mathbb{T}$ respectively. We say that the output streams of R and R' are orderlessly equivalent if, by inputting the same stream S^{in} to the systems, we have $img(S) = img(S')$, where $img(S)$ and $img(S')$ are the images of the corresponding functions.*

Figure 5.2 shows an example of a RETE network and a distributed version of it. Each node of the RETE network is distributed to a cluster of nodes. Moreover, each arc connecting node i to node j is represented by several arcs connecting the nodes in cluster i to some nodes cluster j. For example, the alpha node α_1 in the RETE network is distributed to cluster C_1^{α} and the beta node β_1 is distributed to cluster C_1^{β}.

As both alpha and beta nodes of the RETE-based systems can be modelled by stream functions, the output of a RETE network is a single function S while the output of a

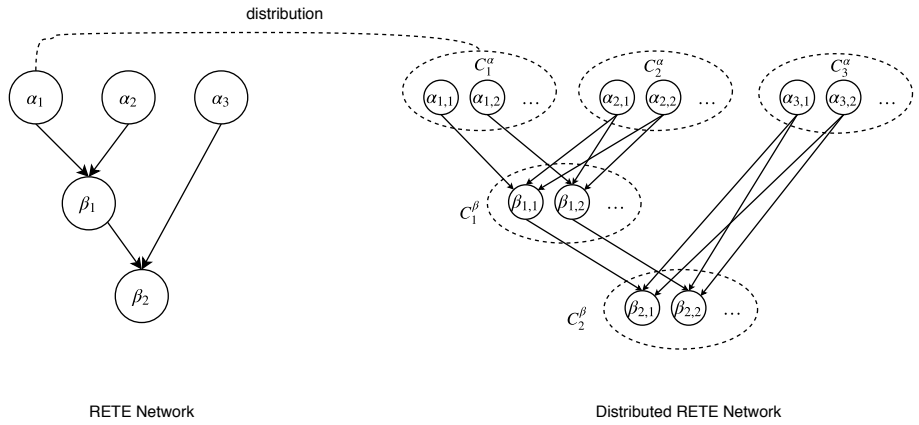


Figure 5.2: Example of A Distributed RETE Network

distributed RETE network consists of multiple functions $S'_j, 1 \leq j \leq n$, where n is the size of the terminal cluster. Consequently, the orderless equivalence between the RETE and the distributed RETE networks becomes the equivalence between the image of function S and the union of images of functions S'_j. In other words, if a tuple t of events reaches a terminal node β_i in a RETE network (thus becomes an element of $img(S)$), the same tuple t is expected to reach one node of the cluster C_i^β in DRESS corresponding to β_i (thus becomes an element of $img(S'_j)$).

Reachability Analysis is a technique widely used in verifying the states of Petri Nets [60, 83, 9]. It can be used to prove the orderless equivalence of two RETE-based systems, as RETE networks are structurally similar and can be easily converted to Petri nets. In the following of this section, we define the reachability of the RETE networks and show how the verification of orderless equivalence of RETE-based systems is carried out using reachability analysis.

5.2.1 Converting RETE Networks to Petri Nets

Recall the construction method of RETE networks. The RETE algorithm compiles each atomic rule r into a triangle shaped network, as shown in Figure 5.3.

The method that converts a RETE network to a Petri net is straightforward:

- For each pair of arcs connecting to a beta node β_i from the nodes j and k, create a

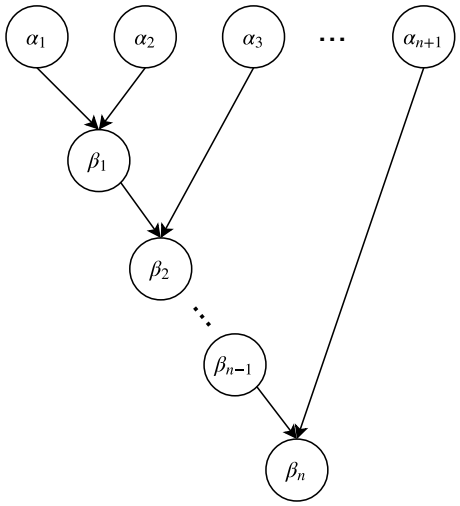


Figure 5.3: A Triangle Shaped RETE Network

transition then replace the arcs with one arc from j to the transition, one arc from k to the transition and another arc from the transition to β_i.

- All created arcs have weight 1.
- Replace all alpha and beta nodes with *places*.

The Petri net converted from the RETE network of Figure 5.3 is shown in figure 5.4. We can now apply reachability analysis techniques for Petri nets on RETE Networks.

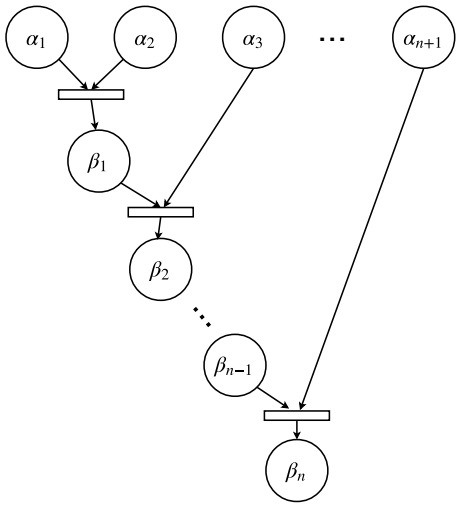


Figure 5.4: RETE Network Represented by Petri Net

5.2.2 State of RETE Networks

A state of the RETE network describes how data items are kept across the nodes. A *token* (denoted by •) is used to indicate an item that reaches and is stored at a certain node. One state can transit to another if it is possible. For example, in *state 1* of Figure 5.5, the condition of beta node β_1 is satisfied (both of its left and right input nodes have valid tokens). Thus a transition to *state 2* is possible where tokens of α_1, α_2 are removed and a token is added to β_1.

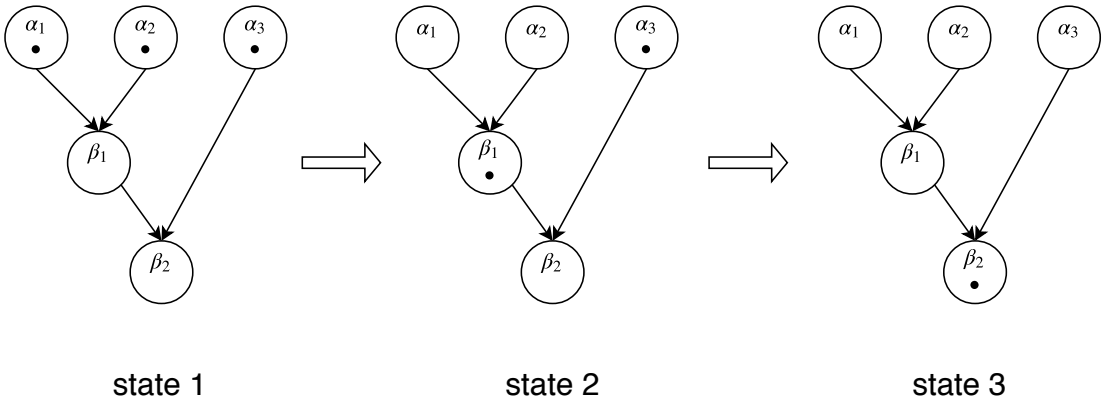


Figure 5.5: State Transition of RETE Networks

A state can be represented by a *marking*. A marking $M = (I_{\alpha_1}, \ldots, I_{\alpha_n}, I_{\beta_1}, \ldots, I_{\beta_m})$ is a set of integers indicating the numbers of tokens at each node of a RETE network with n alpha nodes and m beta nodes. Specifically, I_{α_i} is the number of tokens in alpha node α_i while I_{β_j} is the number of tokens in beta node β_j.

For example, the states of Figure 5.5 can be represented by the marking schema $M = (I_{\alpha_1}, I_{\alpha_2}, I_{\alpha_3}, I_{\beta_1}, I_{\beta_2})$, where states 1,2 and 3 are represented by $M_1 = (1,1,1,0,0), M_2 = (0,0,1,1,0)$ and $M_3 = (0,0,0,0,1)$ respectively. In the remaining of this section, we use the term marking and the term state interchangeably.

Definition 5.2 (transition of marking). *An one-step transition from a state M' to a state M'' inserting a token to node β' is denoted by $M' \overset{\beta'}{\rightsquigarrow} M''$. A marking M'' is reachable from marking M' if there exists a transition sequence that transforms M' to M'', and this is denoted by $M' \rightsquigarrow^* M''$.*

Hence, in the example of Figure 5.5, we have $M_1 \rightsquigarrow M_2 \rightsquigarrow M_3$, and $M_1 \rightsquigarrow^* M_3$

5.2.3 Reachability Graph for RETE Networks

A reachability graph is a directed graph which describes the state transitions of a system. The nodes of a reachability graph correspond to the markings reachable from the initial marking, one node per marking. The arcs are one-step transitions from one marking to another. Figure 5.6 shows an example of two reachability graphs.

Definition 5.3. *Given two reachability graphs G with n markings and G' with m markings, we say that G' simulates G if and only if*

- *G' has more markings than G, i.e. $m > n$ and,*
- *there is one and only one way to divide the markings of G' into n groups $(g_1, \ldots, g_n)$ and associate each of the groups with a unique marking in G, such that for any pair of groups g_i, g_j associated with marking M_i and M_j in G respectively, $M_i \rightsquigarrow M_j \iff \forall M \in g_i \exists M' \in g_j, M \rightsquigarrow M'$.*

For example, in Figure 5.6, the reachability graph G_2 on the right hand side simulates G_1 on the left side, because G_2 has more markings than G_1 and the markings of G_2 can be divided into 5 groups: $g_1 = (M'_0)$, $g_2 = (M'_1)$, $g_3 = (M'_{2,1}, M'_{2,2})$, $g_4 = (M'_{3,1}, M'_{3,2}, M'_{3,3})$, $g_5 = (M'_{4,1}, M'_{4,2})$. Let g_i be associated with M_{i-1}. We can see this is the only way to satisfy the second condition of the reachability graph simulation.

Definition 5.4. *The reachability graph of a RETE network, with respect to the initial marking M_0, is a graph of all reachable markings.*

As the numbers of tokens in alpha nodes are unbounded, in the following diagram the alpha part of the marking is omitted. Figure 5.7 shows the reachability graph for the example in Figure 5.3.

A simplified reachability graph considers only zero or non-zero numbers of tokens, as shown in Figure 5.8.

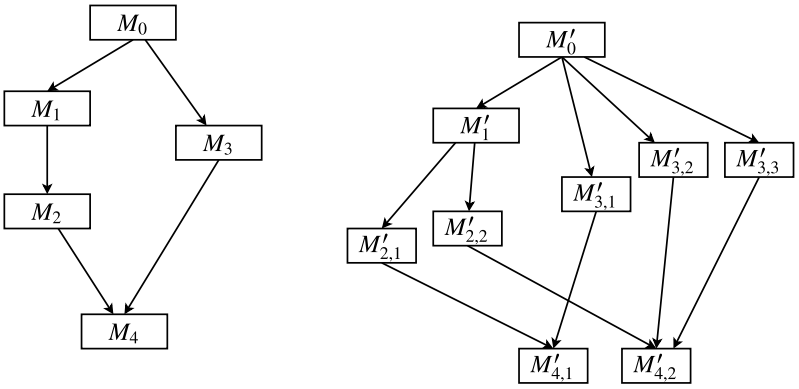


Figure 5.6: Simulation of Reachability Graphs

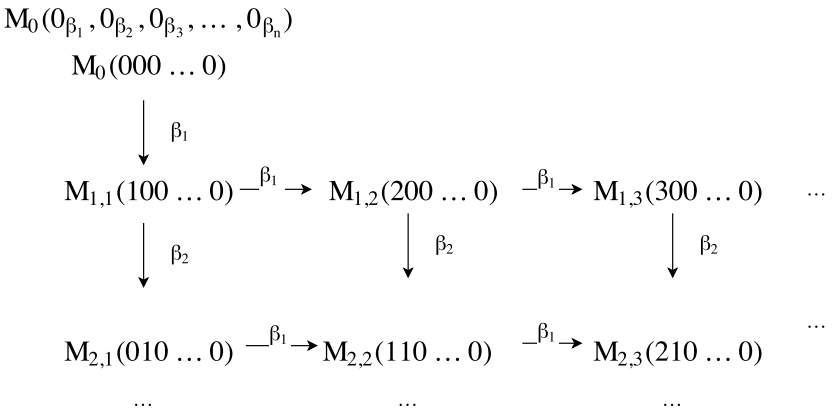


Figure 5.7: Reachability Graph for the RETE Network in Figure 5.3

5.2.4 Reachability Analysis for RETE Networks

Theorem 5.1. *Consider a RETE network R and a distributed version of it R′. Assume reachability graphs G and G′ represent the state transitions of R and R′ respectively. The output streams of R and R′ are orderlessly equivalent if and only if the following conditions hold:*

1. *G′ simulates G.*

2. *Assume R is in a state M_i in G and R′ is in a state M'_i in G′, where the group consisting M'_i is associated with M_i. When the same event e arrives at both systems, if M_i transits to M_j and M'_i transits to M'_j, then the group consisting M'_j is associated with M_j.*

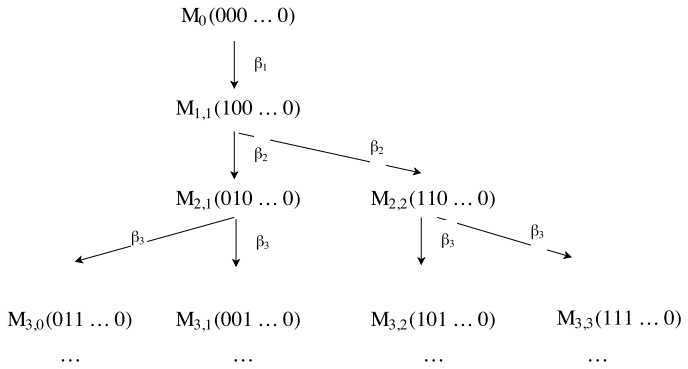


Figure 5.8: Simplified Reachability Graph

3. *Consider marking M_i in G and M'_i in G', where the group containing M'_i is associated with M_i. Consider further a beta node β_j in R which is distributed to the cluster C_j^{β} in R'. If a one-step transition to M_i adds a token to β_j , then a one-step transition to M'_i adds a token to a beta node in the cluster C_j^{β}.*

Proof. **P1:** R and R' both start with their initial markings M_0 and M'_0 respectively. From Definition 5.3, we know that the group containing M'_0 is associated with M_0. Inductively, from the second condition of Theorem 5.1, we know when R is in a state M_y, R' is always in a state M'_y whose group is associated with M_y if both systems are given the same input stream.

P2: Let $M = (I_{\beta_1}, \ldots, I_{\beta_n})$ be the marking schema of G, where I_{β_i} is an integer indicating the number of tokens at node β_i in R. Let β_j be a terminal node. Assume after many transitions, G starts from the initial marking $M_0 = (\ldots, I_{\beta_j} = 0, \ldots)$ and ends up at marking $M_k = (\ldots, I_{\beta_j} \neq 0, \ldots)$, where the value of I_{β_j} is incremented by 1. Then, we have a sequence of one-step transitions $\xi_1 = (M_0 \overset{\beta_{x_1}}{\rightsquigarrow} M_{y_1} \overset{\beta_{x_2}}{\rightsquigarrow} M_{y_2} \rightsquigarrow^* M_k)$.

From **P1**, we know, given the same input stream, R' also has a sequence of one-step transitions $\xi_2 = (M'_0 \overset{\beta'_{x_1}}{\rightsquigarrow} M'_{y_1} \overset{\beta'_{x_2}}{\rightsquigarrow} M'_{y_2} \rightsquigarrow^* M'_k)$. Moreover, the group containing M'_y is associated with M_y. From the third condition of Theorem 5.1, we know for any transitions $\overset{\beta_x}{\rightsquigarrow}$ in ξ_1 resulting a token being added to the beta node β_x in R, there is a transition in ξ_2 resulting a token being added to a beta node β'_x in R'. Also β'_x is in the

cluster corresponding to β_x in R.

P3: As R is a RETE-based system, the conditions presented by all beta nodes along the path from the root to β_j must be satisfied in order to increment I_{β_j}. Hence, there is a sub-sequence $\xi_1' = (\overset{\beta_1}{\rightsquigarrow}, \overset{\beta_2}{\rightsquigarrow}, \ldots, \overset{\beta_j}{\rightsquigarrow})$ of ξ_1 which contains transitions leading to the addition of a token to the terminal node β_j. From **P1** and **P2**, we know there is a sub-sequence of ξ_2 resulting a token being added to a beta node β_j' in R', where β_j' is a node in the cluster distributed from β_j.

P4: From **P1**, **P2** and **P3**, we know for any token that reaches a terminal node of R, there is a same token that reaches a node in a terminal cluster of R'. □

In the following section, we prove that the reachability graphs of the original RETE-based system and DRESS satisfy the assumptions of Theorem 5.1. Hence, we prove that the two systems are orderlessly equivalent.

5.2.5 Reachability of DRESS Networks

Lemma 5.1. *Given a RETE network R and its DRESS representation D, the reachability graphs G^D of D and G^R of R satisfy the conditions of Theorem 5.1.*

Proof. Consider the minimum RETE network with 2 alpha nodes and 1 beta node, and its representation in DRESS, as shown in Figure 5.9. Alpha nodes α_1 and α_2 in the RETE network are distributed to alpha clusters C_1^α and C_2^α in DRESS respectively, and the beta node β_1 is distributed to the beta cluster C_1^β.

Markings of the reachability graph for the minimum RETE network contain only one integer indicating the number of tokens in β_1. The initial marking is $M_0 = (0)$, which transits to $M_1 = (1)$ if the condition of β_1 is satisfied, as shown in the left side of Figure 5.10.

Condition 1: Markings of the reachability graph for the DRESS representation contain n integers for node β_1 (where n is the cluster size). The initial marking is $M_0^d = (0,0,\ldots)$ and there are $2^n - 1$ reachable markings because each integer can be either 0 or

1. Let $g_1 = (M_0^d)$ and $g_2 = (M_{1,1}^d, M_{1,2}^d, M_{1,3}^d)$ (all other reachable markings), and associate g_1, g_2 with M_0,M_1 respectively. From definition 5.3 we know the DRESS reachability graph simulates the RETE reachability graph for the minimum network.

Condition 2: Consider the transition from the initial marking M_0 to M_1 in the RETE reachability graph in Figure 5.10. The transition describes the state change where a token is inserted to the initially empty node β_1 because its condition is satisfied. A one-step transition from the initial marking $M_0^d = (0,0,\dots)$ of DRESS can lead to markings $M_{1,1}^d = (1,0,\dots)$ and $M_{1,2}^d = (0,1,\dots)$ where both belong to the group g_2 which is associated with marking M_1. Similarly, for the transition from M_1 to itself, we can see the condition 2 of Theorem 5.1 is satisfied.

Condition 3: In Figure 5.9, when there are valid tokens in α_1 and α_2, the condition of β_1 is satisfied, which results in a transition from M_0 to M_1 in Figure 5.10 where a token is added to β_1. In DRESS, when there are valid tokens in one of the nodes of the alpha cluster C_1^α and one of the nodes in C_2^α, the condition of one beta node in cluster C_1^β is satisfied hence a token is added to that beta node. This causes a transition from M_0^d to one marking of the group g_2. Similarly, for the transition from M_1 to itself, the condition 3 of Theorem 5.1 holds.

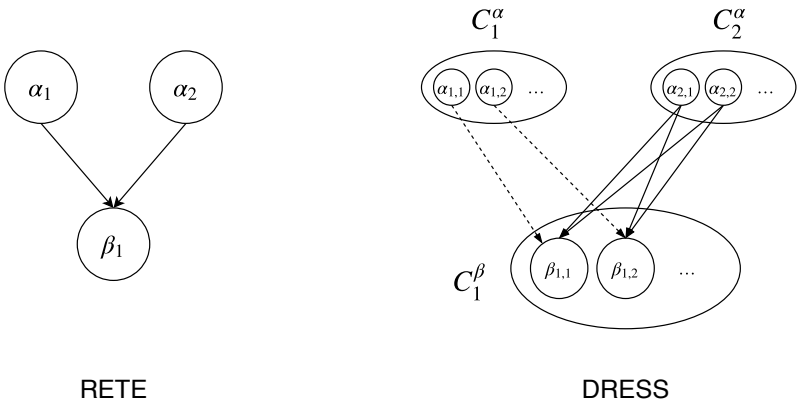


Figure 5.9: Minimum RETE Network and Its DRESS Representation

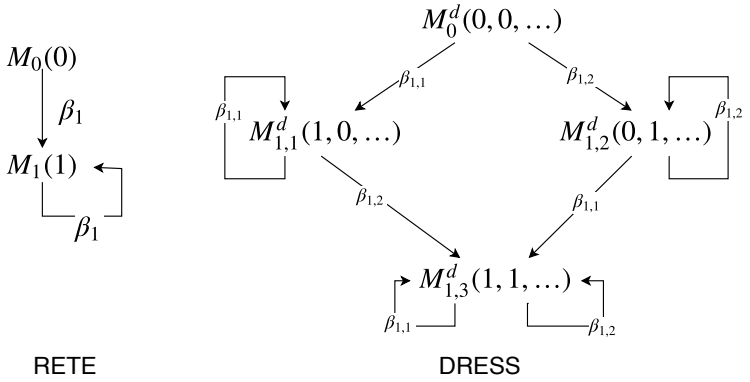


Figure 5.10: Reachability Graphs for Minimum RETE and DRESS Networks

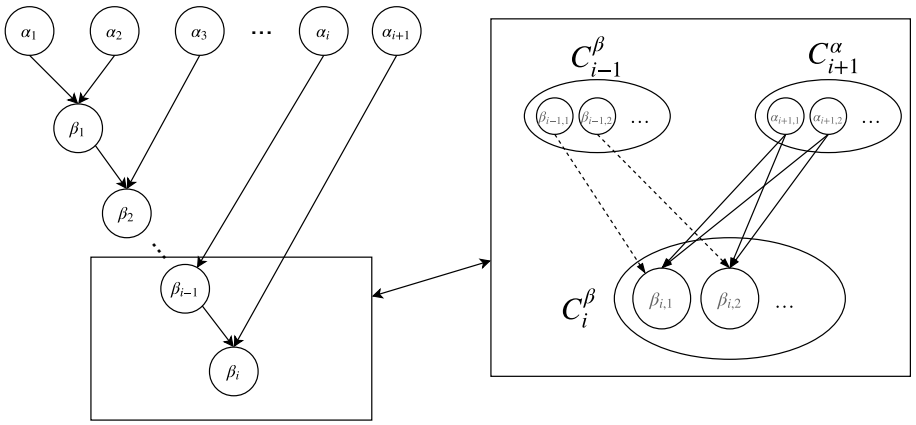


Figure 5.11: RETE Network with i Beta Nodes

Consider a RETE network of $i-1$ beta nodes as shown in Figure 5.11, and assume the conditions of Theorem 5.1 are satisfied for its DRESS representation.

When a new beta node β_i is added to the network, the marking schema M^d of DRESS expands to contain one additional set of integers $I_{\beta_i}^d = (I_{\beta_{i,1}}^d, I_{\beta_{i,2}}^d, I_{\beta_{i,3}}^d)$, yielding

$$M^d = (\ldots, I_{\beta_{i-1}}^d, I_{\beta_i}^d, \ldots).$$

Condition 1: The reachability graph of the RETE network with $i-1$ beta node has $i-1$ markings. For each marking $M_j = (I_{\beta_1}, \ldots, I_{\beta_{i-1}})$ for the RETE reachability graph, we create a group g_j for its DRESS representation and assign all markings $M_j^d = (\ldots, I_{\beta_{i-1}}^d, I_{\beta_i}^d, \ldots)$ to g_j, if M_j^d satisfies $\forall I_{\beta_i} \in M_j, \forall x \in I_{\beta_i}^d : I_{\beta_i} = 0 \iff x = 0$.

The conditions for a transition inserting a token to beta node β_i in the original RETE

network are: 1) there exists valid token(s) in node β_{i-1}, and 2) there exists valid tokens in α_{i+1}.

According to our assumption, (I^d_{i-1}) contains at least one non-zero value, and at least one of the integers of $I^d_{\alpha_{i+1}}$ is non-zero. As the alpha DState is an accumulation of the output of the DCluster, we have a transition in the DRESS reachability graph. Hence, from definition 5.3, the DRESS reachability graph simulates the RETE reachability graph.

Condition 2,3: Consider a transition from $M_i = (\ldots, I_{\beta_{k-1}} \neq 0, I_{\beta_k} = 0, \ldots)$ to $M_j = (\ldots, I_{\beta_{k-1}} \neq 0, I_{\beta_k} \neq 0, \ldots)$ in the RETE reachability graph, where a token is inserted to the beta node β_k. This transition requires a valid token in the beta node β_{k-1} and a valid token in the alpha node α_{k+1}. Assume the DRESS reachability graph transits from $M^d_i = (\ldots, I^d_{\beta_{k-1}}, I^d_{\beta_k}, \ldots)$ to $M^d_j = (\ldots, (I^d_{\beta_{k-1}})', (I^d_{\beta_k})', \ldots)$. We know at least one integer in each of $I^d_{\beta_{k-1}}$ and $I^d_{\alpha_{k+1}}$ is non-zero. Hence a transition is made and one integer of $I^d_{\beta_k}$ will be incremented. Base on how the groups are created above, we know that if M^d_i is associated with M_i then M^d_j is associated with M_j, because I_{β_k} and $I^d_{\beta_k}$ are the only number(s) that have been incremented.

□

Theorem 5.1 and Lemma 5.1 immediately lead to the following corollary.

Corollary 5.1. *Consider a RETE based system R and its DRESS representation D receiving the same input. The output streams of D and R are orderlessly equivalent.*

5.3 The Preservation of Ordering in DRESS Networks

In this section, we will verify that the transformation from a RETE network to a DRESS network preserves the ordering of their output streams. In order to do so, we need to define the ordering first.

Definition 5.5. *Given an output stream of a RETE based system represented by function $S\colon \mathbb{N} \to \mathbb{T}$, we say a tuple $t_i = S(i)$ is placed in S before a tuple $t_j = S(j)$ if and only if*

$i < j$.

$t_i \overset{S}{\prec} t_j$ *denotes this order within the context of S.*

In this section, we focus on the order preservation of RETE based system. Hence, the definition of ordering equivalence needs to be general enough to cover the situations where the output streams contain different set of elements.

Definition 5.6. *Consider two RETE based systems R and R^d receiving the same input stream, yielding the output streams $S^R : \mathbb{N} \to \mathbb{T}$ and $S^d : \mathbb{N} \to \mathbb{T}$, respectively. Let $\mathbb{S}$ be the intersection of the images of S^R and S^d. We say that R^d preserves the output order of R, if* $\forall (t,t') \in \mathbb{S} \times \mathbb{S} : t \overset{S^R}{\prec} t' \iff t \overset{S^d}{\prec} t'$.

Consider the function $S^{\beta_0} : \mathbb{N} \to \mathbb{T}$ presenting the output of the first beta memory ω_0^β of a RETE network adapted from the first alpha memory ω_1^α without further processing. The image of S^{β_0} contains 1-tuples of all elements from the image of the function $S{\restriction}_{A_{p_i}} : A_{p_i} \to \mathbb{E}$. Since A_{p_i} is a subset of $\mathbb{N}$ and $S{\restriction}_{A_{p_i}}$ is a stream function, meaning that the events arrive at ω_1^α in the order of ascending indices, event indices $j \in A_{p_i}$ are mapped to incrementing indices of the first beta memory ω_0^β, as shown in Figure 5.12.

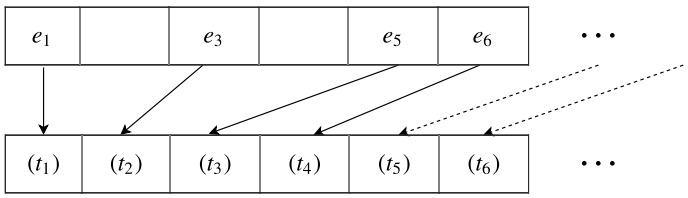


Figure 5.12: Indices of the first beta memory

Therefore, the order of tuples in ω_0^β is defined by the indices of events they contain. In other words, for a given pair of tuples $t_i = S^{\beta_0}(i) = (e_x)$ and $t_j = S^{\beta_0}(j) = (e_y)$, $t_i \overset{S^{\beta_0}}{\prec} t_j \iff i < j \iff x < y$.

We now consider the order of output streams for other beta memories.

Lemma 5.2. *Given a RETE network R and the output stream $S^\beta : \mathbb{N} \to \mathbb{T}$ of any beta memory β in R. Consider a pair of tuples (t_i, t_j), where $t_i = S^\beta(i)$ and $t_j = S^\beta(j)$. Let $\sigma(t)$ be the largest index among all events of the tuple t. Then, $t_i \overset{S^\beta}{\prec} t_j \iff \sigma(t_i) < \sigma(t_j)$.*

Proof. Let $S^{in} : \mathbb{N} \to \mathbb{E}$ be the input stream to R and $\kappa \in \mathbb{N}$ be the largest index of events that the system has received so far.

$$
\begin{aligned}
t_i \overset{S^\beta}{\prec} t_j &\iff i < j && \text{(definition 5.5)} && (5.3)\\
&\iff t_i \text{ arrives at one of the terminal nodes before } t_j && && (5.4)\\
&\iff \exists \kappa : (\forall e_x \in t_i, x \leq \kappa) \wedge (\exists e_y \in t_j, y > \kappa) && && (5.5)\\
&\iff \exists e_y \in t_j : \forall e_x \in t_i, y > x && && (5.6)\\
&\iff \sigma(t_i) < \sigma(t_j) && && (5.7)
\end{aligned}
$$

□

Theorem 5.2. *Consider a RETE network R and its DRESS representation D. If R and D are both given the same input stream, then D preserves the output ordering of R.*

Proof. In DRESS, $x < y$ does not necessarily mean event e_x arrives in the system before event e_y, thus the proof of the statement of lemma 5.2 for DRESS would be problematic. However, the aggregation operators $\uplus^\alpha$ and $\uplus^\beta$ (see section 5.1.2) ensure that tuples t from the stream S^D are output in an order according to $\sigma(t)$.

Let $S^R : \mathbb{N} \to \mathbb{T}$ and $S^D : \mathbb{N} \to \mathbb{T}$ be the output streams of R and D respectively. For any given pair of tuples $t_i = S^R(i)$ and $t_j = S^R(j)$, if there is a pair of tuples $t_x = S^D(x)$ and $t_y = S^D(y)$ such that $t_i = t_x$ and $t_j = t_y$, without loss of generality assume $t_x \overset{S^D}{\prec} t_y$.

$$
\begin{aligned}
t_x \overset{S^D}{\prec} t_y &\iff x < y && \text{(definition 5.5)} && (5.8)\\
&\iff \sigma(t_x) < \sigma(t_y) && \text{(definition of } \uplus^\beta\text{)} && (5.9)\\
&\iff \sigma(t_i) < \sigma(t_j) && (t_i = t_x,\, t_j = t_y) && (5.10)\\
&\iff t_i \overset{S^R}{\prec} t_j && \text{(lemma 5.2)} && (5.11)
\end{aligned}
$$

□

Equation (5.11) indicates the output stream of the DRESS representation of a RETE network preserves the output order. In other words, for any given pair of tuples $(t,t') \in img(S^R) \times img(S^D)$, $t \overset{S^R}{\prec} t' \iff t \overset{S^D}{\prec} t'$.

5.4 Chapter Summary

This chapter begins with a formalisation of the DRESS networks by using the same notations introduced in section 2.1. This is followed by a 2-step verification for the transformation from RETE networks to DRESS networks.

In section 5.2, we prove that the DRESS model transformed from a RETE model produces the same set of outputs as the RETE model, given the same input. This is achieved by reachability analysis. In addition, in section 5.3, we investigate the preservation of the output order in the DRESS model. This opens a possibility to further improve the performance of certain applications of DRESS by removing some constraints related to order preservation in the model, given the application does not concern the output order.

CHAPTER 6

BENCHMARKING DISTRIBUTED RULE ENGINES

This chapter evaluates the proposed model (DRESS) and the automated transformation by a benchmark. It starts with a case study of DRESS in Section 6.1. In Section 6.2, we present a generic benchmark for rule-based event stream processing. Section 6.3 then studies the performance and scalability of DRESS by using the benchmark.

6.1 An Example of A DRESS Application

This section describes an example of a DRESS application based on a simplified version of the banking benchmark[1]. A banking system contains three major classes: *CashFlow*, *AccountingPeriod* and *Account*. Each *Account* object contains the information of one customer. The *AccountingPeriod* contains the starting and ending dates of each accounting period and the *CashFlow* contains the account, the amount and the type (*DEBIT* or *CREDIT*) of a transaction.

This benchmark comes with a data generator and a set of rules that calculate each customer's balance at the end of each *AccountingPeriod*. The rules written in Drools' format are listed below:

```
rule "1 - Credit Cashflow"
  when
```

```
    AccountingPeriod($start:start_date, $end:end_date)
    $flow:Cashflow($account:account,
        date<=$end && >=$start, $amount:amount,type==CREDIT)
  then
    $account.setBalance($account.getBalance()+$amount);
    retract($cashflow);
end
rule "2 - Debit Cashflow"
  when
    AccountingPeriod($start:start_date, $end:end_date)
    $flow:Cashflow($account:account,
        date<=$end && >=$start, $amount:amount,type==DEBIT)
  then
    $account.setBalance($account.getBalance()-$amount);
    retract($cashflow);
end
```

We can observe the propositions, the variable bindings, and the actions from the rules:

```
Propositions:
A: exists AccountingPeriod
B: exists CashFlow and its type is DEBIT
C: exists CashFlow and its type is CREDIT

Variable Bindings:
D: the date of B is between the starting and ending dates of A
E: the date of C is between the starting and ending dates of A

Actions:
F: update balance for corresponding AccountingPeriod
```

Each of the propositions is compiled into an alpha chain that terminates at an alpha memory by the RETE algorithm as described in Chapter 2. In addition, each variable binding is compiled into a beta node and a beta memory. The resulting RETE network for the banking rules is showed in Figure 6.1.

The automated transformer described in Chapter 4 transforms the RETE network into a DRESS network based on the transformation rules. The resulting DRESS network is shown on the right hand side of Figure 6.1.

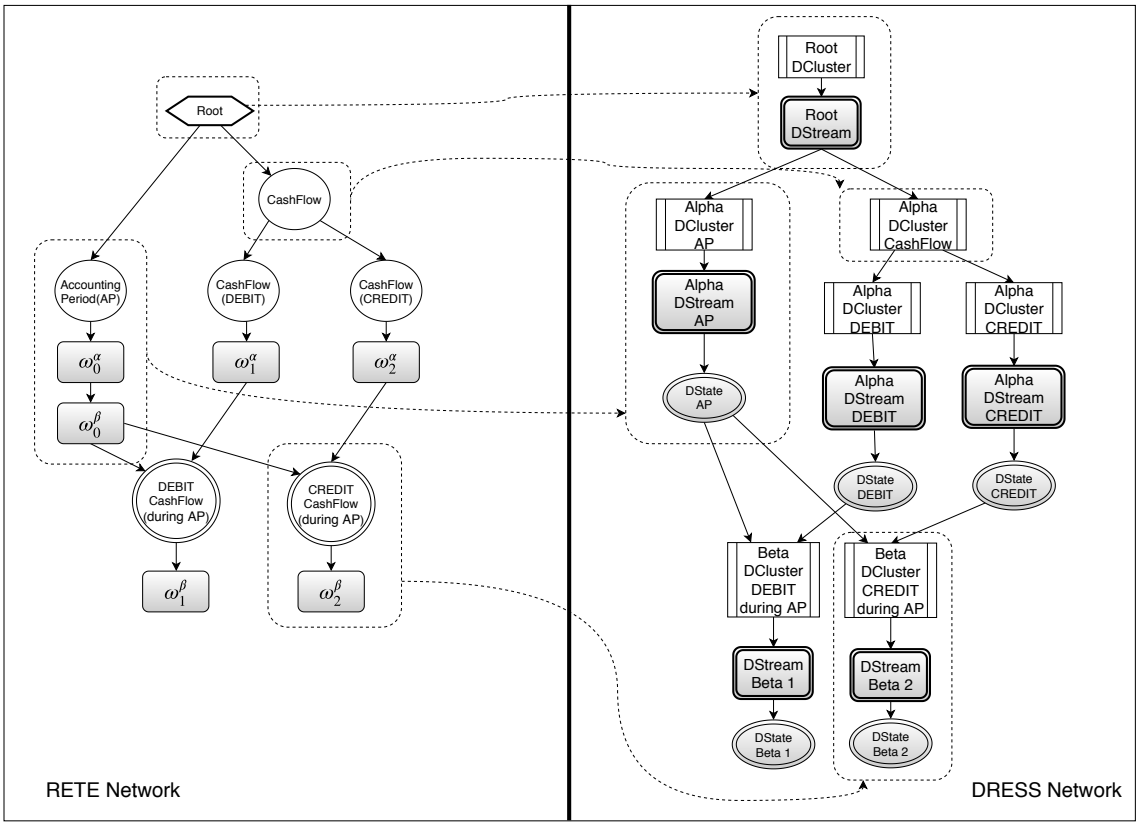


Figure 6.1: RETE and DRESS Networks for the Banking Benchmark

The DRESS network is then executed by DRESS with the Spark Framework. Consider that an input stream is sent to the system. DRESS assigns a set of executors from the root DCluster to convert the format of the events from the input stream. Each of the executors receives a micro batch of the input stream and produces a batch of events with the unified format. All output batches from the root DCluster are aggregated into the root DStream. Afterwards, alpha DClusters are appointed to process the batches from the root DStream. For example, the Alpha DCluster (AccountingPeriod) produces a DStream consisting of batches of AccountingPeriod events, and these batches are stored in the DState (AP). Similarly, CashFlow events with type DEBIT and CREDIT are stored in the DState (DEBIT) and the DState (CREDIT) respectively. Then, beta DClusters creates tuples consisting of DEBIT and CREDIT CashFlow events with their corresponding AccountingPeriods. As both beta DClusters in this example are terminal DClusters, the actions of the rules are performed for every tuple created by the beta DClusters.

This example shows the process of applying DRESS to event stream processing. Firstly, a set of rules is created by domain experts. Then, these rules are compiled by the RETE algorithm into a RETE network. Secondly, the automated transformer transforms the RETE network into a DRESS network which is executed by DRESS with the help of the Spark framework. The event stream is split into micro batches and processed by the DRESS network. Finally, the actions of the rules are performed by the terminal DClusters.

6.2 SONA: A Benchmark for Rule Engines

Many benchmarks [37, 54, 11] have been developed in both industry and academia to evaluate the performance of rule engines. *Miss Manners* [11] is one of the most popular benchmarks which can be used to compare the performances among different rule engines. It is based on the problem of finding an acceptable seating arrangement for guests at a dinner party. The requirement is that each guest is seated next to someone of the opposite sex who shares at least one hobby. This benchmark is designed to stress the beta nodes of a small and simple RETE network. This RETE network is compiled from a set of eight rules based on a depth-first search solution to the problem. In addition, it is estimated that with 128 guests, the rule engine will need to perform several hundred million evaluations.

Although these benchmarks are useful to test certain characteristics of rule engines, there are some issues:

- They usually have a very small number of rules.
- They focus on the worst-case scenarios, which are rarely encountered in real-world applications.
- They ignore streaming applications of rule engines, i.e. they test the performance of rule engines by inputting data into the system all at once.

This section presents the SONA benchmark for rule engines. SONA is a highly configurable benchmark that focuses on the execution time of rule-based event stream processing systems.

SONA is based on a planning problem. A company needs to plant flowers in a group of gardens. Due to different soils in the gardens, each has a different set of requirements for the flowers. Also, for the purpose of variety, there are some requirements on the types of flowers to be planted among the gardens. For example, two gardens next to each other should have flowers of different colours. The company wants to know their options and examine an appropriate sequence of flower data.

This problem can be easily solved by a rule engine with a set of rules. More specifically, we can create a rule for each garden to select suitable types of flowers to be planted, and these rules can be implemented as the alpha nodes of a RETE network. Then, for each of the flower requirements among gardens, we create a rule to join the output streams of two existing rules.

The SONA benchmark consists of a data generator and a configuration. The configuration is a tuple (g, a, b, d, s), where

g is the number of the gardens, which decides the number of *join* (beta) nodes in the network.

a is the number of the requirements each garden has for the flowers to be planted, which decides the number of *filter* (alpha) nodes in each alpha chain.

b is the number of the flower requirements across gardens, which decides the number of beta nodes with a variable binding.

d is the size of the data set, i.e. the number of flowers.

s is the amount of flower data generated per second.

SONA is generic and highly configurable, which means that, by adjusting the variables of the configuration, it is able to stress some or all aspects of the rule engines. For RETE

based systems, the numbers of alpha nodes and beta nodes scale with the configuration variables a and g, respectively. In addition, the data size scales with the variable d and variable s controls the speed of the data generation.

6.3 Evaluating DRESS with SONA

In this section, we study the performance of DRESS by using the SONA benchmark. In Section 6.3.1, we show how the RETE network from a SONA configuration is created and transformed into a corresponding DRESS network. Section 6.3.2 discusses the performance of DRESS.

6.3.1 Experiment Setup

The Rules

To illustrate the use of the SONA benchmark with DRESS, a minimal configuration $(g = 3, a = 2, b = 1, d, s)$ can be employed. This configuration results in a problem consisting of three gardens. Each garden has two requirements for the flowers and there is one addition requirement on the types of flowers to be planted between two gardens.

Rules written in the Drools' format can be created for this configuration. The variable $a = 2$ controls the number of flower requirements each garden has. Hence, for each of the three gardens, a rule with two propositions is created. For example, for garden g_1, we have:

```
rule "Select suitable flowers for garden g1"
when
    $f : Flower( property1=requirement1, property2=requirement2 )
then
    Tuple t = new Tuple()
    t.first = "g1"
    t.second.put("g1", f)
    insert(t)
end
```

This rule selects all flowers satisfying `requirement1` and `requirement2` of the garden g_1 and inserts a tuple to the system. The second element of this tuple is a map from gardens to selected flowers (with current rules, it contains one garden), and the first element is a symbol representing the list of gardens these flowers can be planted into.

To join the results for the two gardens g_1 and g_2 with a variable binding, we can create a rule which selects two tuples satisfying the requirements of g_1 and g_2 and creates a variable binding:

```
rule "Select suitable flowers for garden g1&g2 with a variable binding"
when
    $t1 : Tuple( first="g1" )
    $t2 : Tuple( first="g2" )
    t1.second.get("g1").property1 == t2.second.get("g2").property2
then
    Tuple t = new Tuple()
    t.first = "g1&g2"
    t.second.putall(t1.second)
    t.second.putall(t2.second)
    insert(t)
end
```

This rule requires the variable `property1` of flowers in garden g_1 to be equal to the variable `property2` of flowers in garden g_2.

Then another rule can be created to join the flowers suitable for garden g_3 and the pairs of flowers suitable for g_1 and g_2.

```
rule "Select suitable flowers for garden g1&g2&g3"
when
    $t1 : Tuple( first="g1&g2" )
    $t2 : Tuple( first="g3" )
then
    Tuple t = new Tuple()
    t.first = "g1&g2&g3"
    t.second.putall(t1.second)
    t.second.putall(t2.second)
    insert(t)
end
```

Finally, for every found solution to the problem, a rule is created to perform the action:

```
rule "Action for found flowers suitable for garden g1&g2&g3"
when
    $t : Tuple( first="g1&g2&g3" )
then
    // action.
end
```

The RETE Network

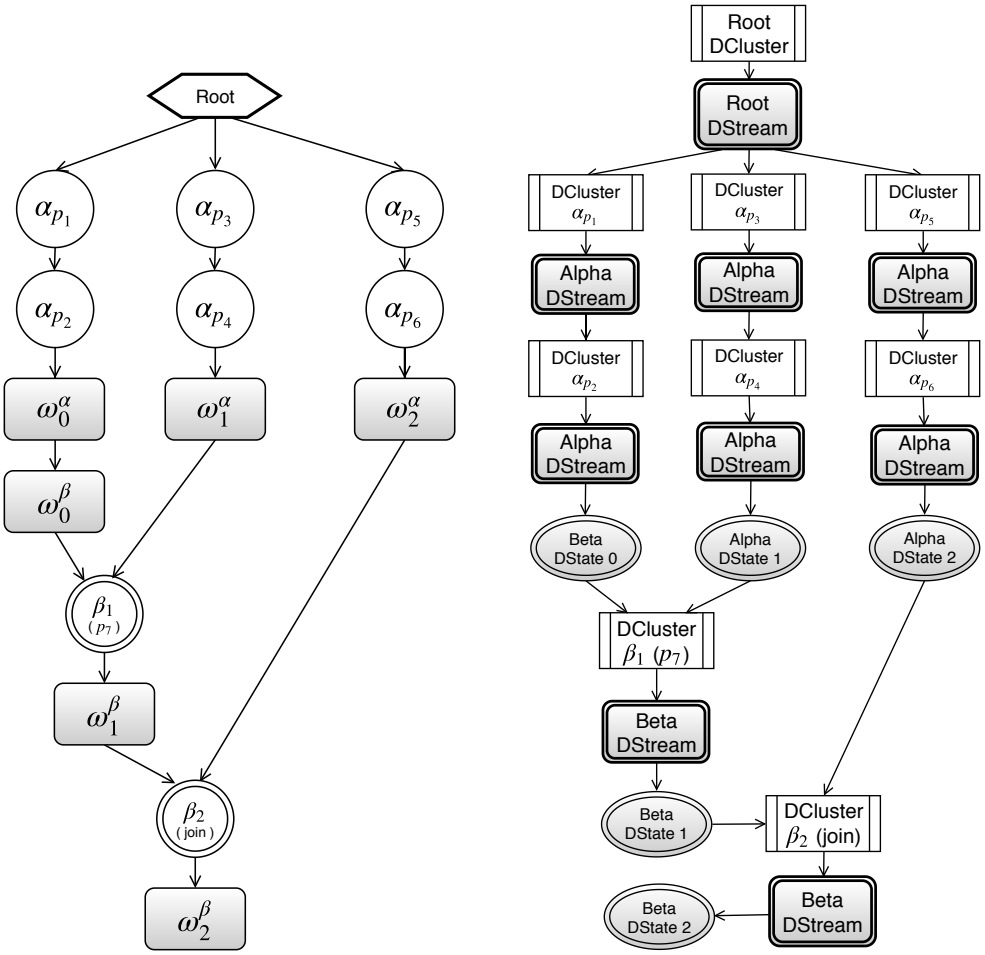
Depending on different optimisation strategies, the above rules may be compiled into several RETE networks, e.g. one RETE network for each rule. The terminal nodes of these RETE networks add new facts, which are defined by the **action** part of the rules, into the system. Adding new facts introduces extra time for transferring the data, which distracts the effort to test the performance of the engine. In order to avoid this distraction, we merge the rules into a single rule as follows:

```
rule "Select suitable flowers for garden g1&g2&g3"
when
    $f1 : Flower( <g1: proposition 1>, <g1: proposition 2> )
    $f2 : Flower( <g2: proposition 3>, <g2: proposition 4> )
    $f3 : Flower( <g3: proposition 5>, <g3: proposition 6> )
    <g1&g2: proposition 7>
then
    // action.
end
```

The patterns f_1, f_2 and f_3 are compiled into alpha chains by the RETE algorithm. Each of the chains has two alpha nodes compiled from the corresponding propositions of the patterns. Moreover, the variable binding (proposition 7) is represented by the beta node β_1, which selects elements that satisfy the variable binding from the alpha chains of pattern $f1$ and pattern $f2$. Finally, the beta node β_2 joins the outputs of β_1 and the alpha chain of f_3. The resulting RETE network is shown in Figure 6.2.a.

Remark. *The following can be observed from Figure 6.2.(a). Given a SONA configuration* (g,a,b,d,s)*, the generated RETE network contains* $\mathbf{g} \times \mathbf{a}$ *alpha nodes, which form* $\mathbf{g}$ *alpha chains. The number of beta nodes is* $\mathbf{g}-1$*. In addition,* $\mathbf{b}$ *of the beta nodes are*

variable binding nodes and the rest are join nodes.



(a) RETE Network (b) DRESS Network

Figure 6.2: RETE and DRESS Networks for the SONA Benchmark

The DRESS network

As previously established, the RETE network representing the rule can be automatically transformed into a DRESS network. The root node, alpha nodes and beta nodes are transformed into corresponding `DClusters`, each of which has an output `DStream`. The alpha and beta memories are transformed into `DStates`. This transformation results in a DRESS network, as shown in Figure 6.2.(b). The target Spark code can be found in appendix B.

Afterwards, this DRESS network is executed on the Spark platform. In the following

section, we evaluate the performance of DRESS with different SONA configurations.

6.3.2 The Performance of DRESS

This section conducts three experiments to investigate DRESS in regards to the aspects of capability, performance and scalability. The first two experiments set the variable d and s of the SONA configurations to the same value, which means the data is inserted into the system at once. The third experiment evaluates DRESS from the streaming point of view, by adjusting the value of the variable s.

Experiment 1: Comparison of DRESS and Drools

The first experiment compares DRESS with Drools in order to evaluate the capability of DRESS. Due to the limitations of Drools, this experiment was conducted on a single machine with the following hardware specifications:

CPU	2.2GHz 6-core Intel Core i7
Memory	16 GB 2400MHz DDR4
Disk	256GB SSD
Java Version	1.8.0_60
JVM	Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)

As of version 6.2.0, the execution engine of Drools runs on a single thread. For fair competition, the number of executors for DRESS is set to 1. In addition, the maximum memory is set to 8GB for both systems.

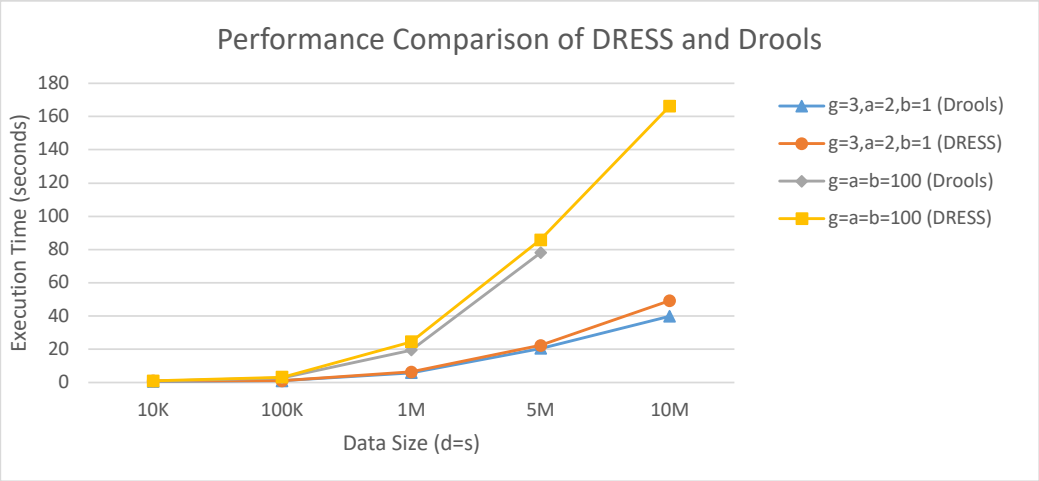


Figure 6.3: Performance Comparison of DRESS and Drools

Two SONA configurations are used in this experiment, including the one described in section 6.3.1. For the other configuration, the values of the variables g, a and b are all set to a hundred, which means the numbers of gardens, alpha chains, nodes in each alpha chain and beta nodes are set to a hundred. Moreover, the variables d and s are set to an equal value which scales up to 10 million. These two configurations and the five different values (10K, 100K, 1M, 5M, 10M) of d and s make up ten tests for both systems. Each test was performed a minimal 2 times and the average execution time was recorded. If the difference between the test results exceeds 10%, further tests were performed until confidence was reached.[1] The results are shown in Figure 6.3.

As can be seen in the results, DRESS and Drools have a similar performance on a single machine. More specifically, Drools is slightly faster with both configurations. A possible reason for why Drools outperforms in this experiment is that DRESS has the overhead of managing the cluster and scheduling jobs, even though there is only one node in this cluster.

It is important to note that there is no result for Drools with configuration $(g = a = b = 100, d = s = 10\text{M})$, while DRESS completed the test in around 166 seconds. This is because Drools crashed due to a memory explosion. Although we can enable Drools to

[1]We define the confidence as the average of the absolute differences (in percentage) to the mean value does not exceed 10%.

work on larger data sizes by installing more memory, it is clear that its memory model is less flexible in terms of garbage collection, which makes it incapable of processing large data sets.

This experiment shows that DRESS is capable of processing larger data sets than Drools even on a single machine. It also shows that DRESS can provide a comparable performance to Drools.

Experiment 2: Scalability of DRESS

This experiment evaluates the scalability of DRESS by running the SONA benchmark on a fully distributed environment. The variables g, a and b are set to a hundred, and the variables d and s are set to the same value, which scales up to 100 million. This experiment was conducted on an Amazon Web Services (AWS) cluster with 12 nodes. Each of these nodes has two vCPUs (cores) and 3.75 GB memory, which makes up a total of 24 executors for Spark. These executors are running on 2.9 GHz Intel Xeon E5-2666 v3 CPUs and the software specifications remain the same as experiment 1. In addition, two nodes of the same specifications were used for the Kafka message queue.

Five tests with different numbers (4,8,12,16,24) of available executors were performed in order to evaluate how DRESS scales with more computing resources. The results are shown in Figure 6.4.

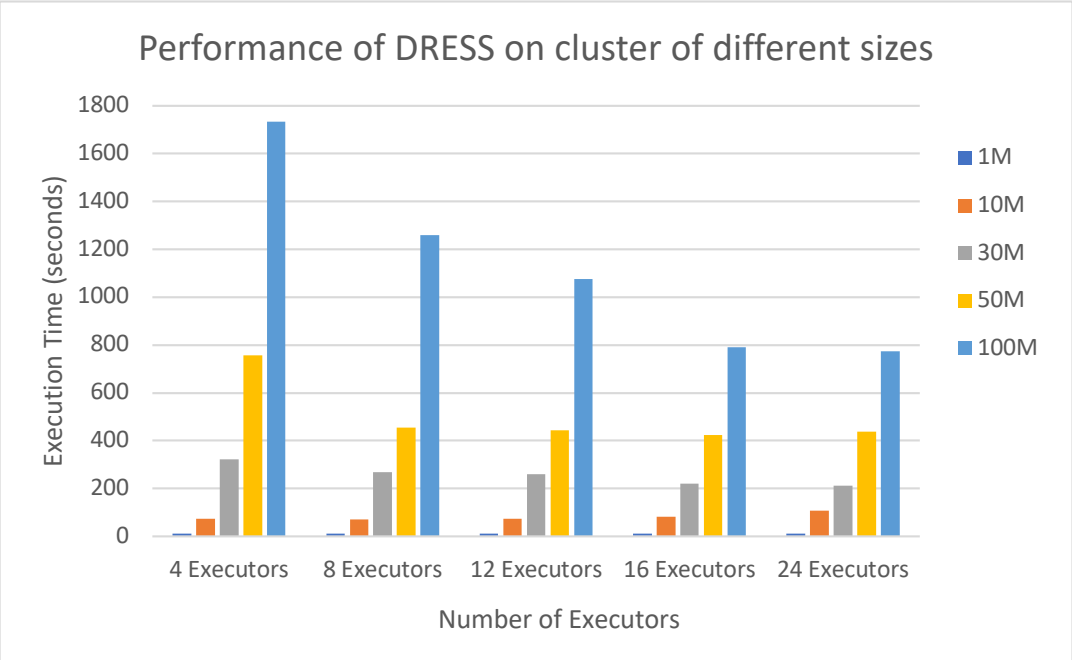


Figure 6.4: Performance of DRESS on clusters of different sizes

As shown, the number of executors has a minimal impact on the execution times of DRESS with smaller data sets (1M and 10M). Specially, for the 10M data set, 24 executors took slightly more time to complete the test than 4 executors. This is due to the fact that the overhead of managing 24 executors exceeded the time needed for the processing.

When the data size goes over 30M, the execution time declines with more executors. For example, the employment of 24 executors halved the execution time that 4 executors needed for the 100M data set. It can also be observed from the results that, for the 100M data set, the execution time declines dramatically from 4 executors to 16. However, after adding 8 more, 24 executors did not bring further noticeable improvement.

This experiment has drawn two major conclusions, one of which is that the performance of DRESS with large data sets can be improved by adding more computing resources (executors). This shows the scalability of DRESS for large event stream processing.

The other conclusion is that there is a limit to such improvements, and when the limit is reached, adding more executors will not reduce the execution time. This limit is a result of several factors. First, stateful computation over the input streams cannot be fully

parallelised. More specifically, there are event sequences in which the processing of one event relies on the processing of another. If the input stream as a whole is one such sequence, the improvement brought by parallelisation will be minimal. Second, when the time needed to process the data on a cluster gets close to the overhead of cluster management and data transferring, adding more nodes will not contribute to the effort of parallelisation.

Experiment 3: Performance of DRESS for Event Stream Processing

The third experiment evaluate the performance of DRESS specially for applications of event stream processing. The same hardware environment of experiment 2 is used and the SONA configuration is fixed to $(g = a = b = 100, d = 100\text{M}, s = 1\text{M})$. This experiment focuses on the response time of DRESS when dealing with large data sets. The response time refers to the time duration from the moment of receiving one event by the system to the moment of outputting results corresponding to that event. The maximum and average response times for this test are recorded, as shown in Figure 6.5.

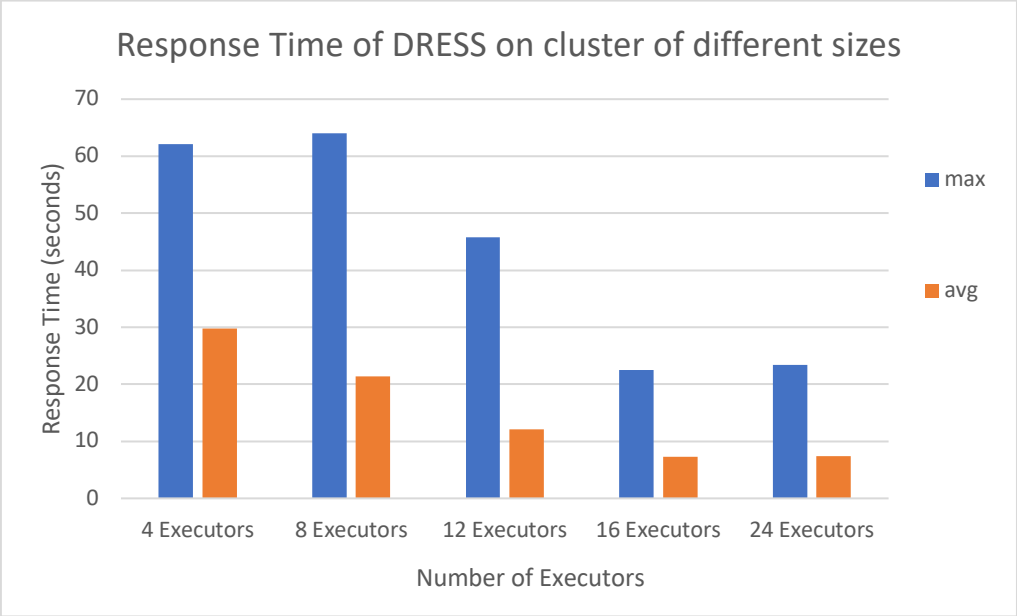


Figure 6.5: Response Time of DRESS on clusters of different sizes

As figure 6.5 shows, the maximum response time of 4 and 8 executors is above 60

seconds. It drops to 46 seconds with 12 executors, and drops further to 22 seconds with 16 and 24 executors. The maximum response time of the system usually happens as a result of a task failure, e.g. an incoming event is corrupted during the network transfer. The results indicate that, with fewer executors, the system takes longer to find available executors in the case of a node failure, as other nodes are most likely occupied by other tasks.

The average response time improves steadily from 30 seconds with 4 executors to 8 seconds with 16 executors. There is no further improvement from 16 to 24 executors. This is due to the same limit reflected in the second conclusion of experiment 2.

Discussion

These experiments show the capability, performance and scalability of DRESS.

In experiment 1, DRESS was able to process larger data sets than Drools, as a benefit of building the rule engine with a general purpose stream processing paradigm which excels in memory management.

Experiment 2 and 3 demonstrated the scalability and performance of DRESS in the processing of large data sets on distributed clusters of different sizes. The results show that DRESS reduces both the execution and response time by adding more computing resources. However, they also show that there is a ceiling of the improvement brought by distribution. This is due to the stateful computation and the overhead of managing the clusters, although the latter had a relatively smaller impact. More specifically, the improvement relies heavily on the portion of data that requires stateful computation.

Some research has been conducted to parallelise stateful computations[28, 24, 87]. For example, [24] studied and introduced a set of state access patterns for managing access to states in parallel computations over streams. The authors identified some cases in which there are clearly defined and restricted state updates and parallelism can be exploited because of the restrictions. However, these works are still in the early stages and can only provide limited parallelism. Hence, high level parallelisation for stateful

computations remains an open research question.

6.4 Chapter Summary

This chapter evaluates the performance of the proposed approach (DRESS).

Section 6.2 presents SONA (a benchmark for rule based event stream processing) as a scalable and configurable alternative to current evaluation frameworks for rule engines.

Section 6.3 begins with a minimal example, showing how SONA is used in evaluating the performance of DRESS, as well as the automated transformation process. This is followed by experiments with different configurations of SONA. The results are presented and discussed in section 6.3.2.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This chapter concludes the thesis by summarising the work presented in this thesis in section 7.1. In addition, section 7.2 discusses some aspects of the proposed model, which can be further improved, and points out the directions for future research.

7.1 Summary of the Thesis

The main contribution of this thesis is the presented rule-based architecture for large event stream processing. This architecture aims at improving current rule-based methods by removing inherent load imbalances from the rules and the event streams. This is achieved by applying dynamic job assignment and micro-batching techniques to rule engines.

The presented architecture (DRESS) is built on top of the Spark Streaming framework and YARN. Specifically, it avails the facilities provided by Spark Streaming for micro-batching based event processing, i.e. DStream and RDD. Moreover, it uses YARN for its ability to dynamically assign jobs based on data locality. The advantage of the presented architecture is that the load imbalances introduced by the rules are removed, i.e. some rules may be more 'popular' than others thus they have more workloads. In addition, this architecture also removes the imbalances inherited from the input streams, i.e. the scenario in which the workload of a particular node changes over time.

In chapter 4, an automated transformation from current rule-based models to DRESS

is presented. This enables current RETE based rule engines to automatically transform to the DRESS rule engine. It is worth to note that because the transformation is based on MDA and meta-modelling, models that are not based on RETE can also be transformed to DRESS, as long as a mapping between their meta-models and the DRESS meta-model can be found.

Chapter 5 proves the correctness of the proposed architecture with the help of the reachability analysis technique. It also investigates the preservation of the output order in the DRESS architecture. The separation of the output ordering from the correctness proof indicates that for some applications, the distributed model can be further improved by loosing the constrictions that are related to order preservation.

Chapter 6 proposes a generic and configurable benchmark for distributed rule-based event stream processing systems. This benchmark is used to evaluate the performance of DRESS. By analysing the results of the evaluation, some advantages and weakness of DRESS are revealed.

7.2 Future Work

Following the contributions made by this thesis, a number of directions may lead to further improvement of rule-based event stream processing.

As this work focuses on improving the performance of current techniques, the RETE model used in this research is a subset of the original RETE model. For example, it does not support <NOT> nodes that are compiled from negated condition elements of the rules. Moreover, the original RETE algorithm, as described by Forgy [31], allows two kinds of changes to the working memory: adding an element (fact) and deleting an element. However, the DRESS architecture does not support the removal of existing facts from the network. These remain tasks for future work.

Another direction for future research would be the introduction of different optimisations to DRESS. For example, the PHREAK algorithm [67], which is an evolution of the

RETE algorithm, has made various improvements to the way the network of nodes is evaluated. These improvements, such as *lazy rule evaluation* may be brought into DRESS.

Other future work would involve the addition of temporal constraints to the proposed model. Current rule engines keep all facts in their networks; for example, a RETE network stores all received facts in its working memories and a DRESS network stores all facts in its DStates. Over time, this requires a lot of resources and slows down the computation. Some proposals [82, 81, 51] have been found in the literature which remove outdated facts by adding temporal constraints to the rules.

Forward chaining is widely used in implementing rule engines because, traditionally, the data sets are relatively small. Its main objective is to optimise the beta networks. By storing all partially matched patterns in the memory, the processing time of the beta networks is vastly reduced (i.e. time-space tradeoff). This is problematic with larger data sets, as there are patterns that will probably never be fully matched and they will nevertheless introduce unnecessary computations. Furthermore, as the data sets getting bigger and bigger, the system may become vulnerable to memory explosion. On the other hand, in the goal-driven backward chaining approach, the number of goals is manageable and the most time-consuming work is to match the goals to the data sets. Since there is no state involved, this matching can be highly parallelised. Therefore, backward chaining rule engines built with big data technologies may be a future research topic.

APPENDIX A

SITRA RULES FOR TRANSFORMING RETE NETWORKS TO DRESS NETWORKS

A.1 Complete Transformation Rules

RootNode to RootDCluster

```
package uk.ac.bham.dress;
import uk.ac.bham.dress.models.dress.*;
import uk.ac.bham.dress.models.rete.*;
import uk.ac.bham.sitra.*;

public class RuleRootNode implements Rule<RootNode, RootDCluster>{

    public boolean check(RootNode source) {
        return true;
    }

    public RootDCluster build(RootNode source, Transformer t) {
        RootDCluster rootDCluster = new RootDCluster("root");
        DStream dStream_root = new DStream("root");
        rootDCluster.addChild(dStream_root);
        return rootDCluster;
    }

    public void setProperties(RootDCluster target,
                              RootNode source, Transformer t) {
        for(Object node: t.transformAll(source.getChildren())){
            AlphaDCluster alphaDCluster = (AlphaDCluster) node;
            target.getChildren().get(0).addChild(alphaDCluster);
        }
    }
}
```

AlphaNode to AlphaDCluster

```
public class RuleAlphaNode implements Rule<AlphaNode, AlphaDCluster>{

    public boolean check(AlphaNode source) {
        return true;
    }

    public AlphaDCluster build(AlphaNode source, Transformer t) {
        AlphaDCluster alphaDCluster=new AlphaDCluster(source.getIdentity());
        DStream dStream = new DStream(source.getIdentity());
        alphaDCluster.addChild(dStream);
        return alphaDCluster;
    }

    public void setProperties(AlphaDCluster target,
                              AlphaNode source, Transformer t) {
        for(Object node: t.transformAll(source.getChildren())){
            Node child = (Node) node;
            target.getChildren().get(0).addChild(child);
        }
    }
}
```

BetaNode to BetaDCluster

```
public class RuleBetaNode implements Rule<BetaNode, BetaDCluster>{

    public boolean check(BetaNode source) {
        return true;
    }

    public BetaDCluster build(BetaNode source, Transformer t) {
        BetaDCluster betaDCluster = new BetaDCluster(source.getIdentity());
        DStream dStream = new DStream(source.getIdentity());
        betaDCluster.addChild(dStream);
        return betaDCluster;
    }

    public void setProperties(BetaDCluster target,
                              BetaNode source, Transformer t) {
        for(Object node: t.transformAll(source.getChildren())){
            Node child = (Node) node;
            target.getChildren().get(0).addChild(child);
        }
    }
}
```

Working Memories to DState

```
public class RuleMemory implements Rule<Memory, DState>{

    public boolean check(Memory source) {
        return true;
    }

    public DState build(Memory source, Transformer t) {
        DState dState = new DState(source.getIdentity());
        return dState;
    }

    public void setProperties(DState target,
                              Memory source, Transformer t) {
        for(Object node: t.transformAll(source.getChildren())){
            Node child = (Node) node;
            target.addChild(child);
        }
    }

}
```

A.2 The Transformer

```
public class Rete2DressTransformer {

    public static RootDCluster transform(RootNode rete){
        List<Class<? extends Rule<?, ?>>> rules = new ArrayList<>();

        rules.add(RuleRootNode.class);
        rules.add(RuleAlphaNode.class);
        rules.add(RuleBetaNode.class);
        rules.add(RuleAlphaMemory.class);
        rules.add(RuleBetaMemory.class);


        Transformer t = new SimpleTransformerImpl(rules);
        RootDCluster rootDCluster = (RootDCluster) t.transform(rete);
        return rootDCluster;
    }
}
```

APPENDIX B

SPARK CODE FOR SONA BENCHMARK

B.1 Configuration $(g = 3, a = 2, b = 1)$

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils

sc = SparkContext(appName="dress")
sc.setLogLevel("ERROR")

ssc = StreamingContext(sc, 1)
ssc.checkpoint("hdfs://hdfs-server:3181/dress_checkpoint")


def updateStateFunc(newState, oldState):
    if oldState is None:
        oldState = {}
    for state in newState:
        oldState = state or oldState
    return oldState


# Kafka message queue
kafkaStream = KafkaUtils.createStream(ssc,\
    'kafka-server:2181', 'dress', {'dress':1})

# RootDCluster
rootDStream = kafkaStream\
             .map(lambda v: eval(v[1]))\
             .map(lambda v: (str(v['id']), v))

# Alpha DCluster
# Patterns
```

```
# f1
alphaDStream_f1_p1 = rootDStream\
                     .filter(lambda x: x[1]['p1'] == "true")
alphaDStream_f1_p2 = alphaDStream_f1_p1\
                     .filter(lambda x: x[1]['p2'] == "true")
alphaDState_f1 = alphaDStream_f1_p2\
                .updateStateByKey(updateStateFunc)

# f2

alphaDStream_f2_p3 = rootDStream\
                     .filter(lambda x: x[1]['p3'] == "true")
alphaDStream_f2_p4 = alphaDStream_f2_p3\
                     .filter(lambda x: x[1]['p4'] == "true")
alphaDState_f2 = alphaDStream_f2_p4\
                .updateStateByKey(updateStateFunc)


# f3

alphaDStream_f3_p5 = rootDStream\
                     .filter(lambda x: x[1]['p5'] == "true")
alphaDStream_f3_p6 = alphaDStream_f3_p5\
                     .filter(lambda x: x[1]['p6'] == "true")
alphaDState_f3 = alphaDStream_f3_p6\
                .updateStateByKey(updateStateFunc)

# Beta DCluster

# f1,f2, variable bindings: p7

def variable_binding_f1_f2_p7(rdd1, rdd2):
    pairs = rdd1.cartesian(rdd2)\
                .filter(lambda v: v[1][0]['vb_1'] == v[1][1]['vb_1'])
    return pairs

betaDStream_f1_f2_left = alphaDStream_f1_p2\
                         .transformWith(variable_binding_f1_f2_p7\
                         ,alphaDState_f2)
betaDStream_f1_f2_right = alphaDStream_f2_p4\
                          .transformWith(variable_binding_f1_f2_p7\
                          ,alphaDState_f1)
betaDStream_f1_f2 = betaDStream_f1_f2_left\
                    .union(betaDStream_f1_f2_right)
betaDState_f1_f2 = betaDStream_f1_f2\
                   .updateStateByKey(updateStateFunc)


# f1,f2,f3, variable bindings: None
def join_f1_f2_f3(rdd1, rdd2):
    pairs = rdd1.cartesian(rdd2)
```

```
    return pairs
betaDStream_f1_f2_f3_left = betaDStream_f1_f2\
                           .transformWith(join_f1_f2_f3\
                           ,alphaDState_f3)
betaDStream_f1_f2_f3_right = alphaDStream_f3_p6\
                            .transformWith(join_f1_f2_f3\
                            ,betaDState_f1_f2)
betaDStream_f1_f2_f3 = betaDStream_f1_f2_f3_left\
                      .union(betaDStream_f1_f2_f3_right)
betaDState_f1_f2_f3 = betaDStream_f1_f2_f3\
                     .updateStateByKey(updateStateFunc)

# actions
def action_h1(rdd):
    rdd.pprint()

betaDStream_f1_f2_f3.foreachRDD( action_h1 )


ssc.start()
ssc.awaitTermination()
```

APPENDIX

LIST OF REFERENCES

[1] (2009) A benchmark for rule engines based on a banking system. URL `https://github.com/codehaus/rulessandpit`.

[2] Akehurst, D.H., Bordbar, B., Evans, M.J. et al. (2006) "Sitra: Simple transformations in java." In **International Conference on Model Driven Engineering Languages and Systems**. Springer. pp. 351–364

[3] Akerkar, R. and Sajja, P. (2010) **Knowledge-based systems**. Jones & Bartlett Publishers.

[4] Akidau, T., Balikov, A., Bekiroğlu, K. et al. (2013) Millwheel: fault-tolerant stream processing at internet scale. **Proceedings of the VLDB Endowment**, 6 (11): 1033–1044

[5] Anicic, D., Fodor, P., Rudolph, S. et al. (2010) "A rule-based language for complex event processing and reasoning." In **International Conference on Web Reasoning and Rule Systems**. Springer. pp. 42–57

[6] Aref, M.M. and Tayyib, M.A. (1998) Lana–match algorithm: a parallel version of the rete–match algorithm. **Parallel Computing**, 24 (5-6): 763–775

[7] Batory, D. (1994) **The LEAPS algorithms**. Univ., Department of Computer Sciences.

[8] Bergmann, G., Dávid, I., Hegedüs, Á. et al. (2015) "Viatra 3: a reactive model transformation platform." In **International Conference on Theory and Practice of Model Transformations**. Springer. pp. 101–110

[9] Berthomieu, B. and Diaz, M. (1991) Modeling and verification of time dependent systems using time petri nets. **IEEE transactions on software engineering**, 17 (3): 259–273

[10] Bowles, J., Alwanain, M., Bordbar, B. and Chen, Y. (2014) "Matching and merging scenarios automatically with alloy." In **International Conference on Model-Driven Engineering and Software Development**. Springer. pp. 100–116

[11] Brant, D.A., Grose, T., Lofaso, B.J. and Miranker, D.P. (1991) "Effects of database size on rule system performance: Five case studies." In **VLDB**. vol. 91, pp. 287–296

[12] Buchanan, B. (1984) Rule based expert systems. **The MYCIN Experiments of the Stanford Heuristic Programming Project**.

[13] Buchanan, B.G. and Duda, R.O. (1983) "Principles of rule-based expert systems." In **Advances in computers**. Elsevier. vol. 22, pp. 163–216

[14] Cabrera, F., Copeland, G., Freund, T. et al. (2002) Web services coordination (ws-coordination). **joint specification by BEA, IBM, and Microsoft**.

[15] Carbone, P., Katsifodimos, A., Ewen, S. et al. (2015) Apache flink: Stream and batch processing in a single engine. **Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**, 36 (4).

[16] CHANG, T.C. and Terwilliger Jr, J. (1987) A rule based system for printed wiring assembly process planning. **International Journal of Production Research**, 25 (10): 1465–1482

[17] Chauvel, F. and Jézéquel, J.M. (2005) "Code generation from uml models with semantic variation points." In **International Conference on Model Driven Engineering Languages and Systems**. Springer. pp. 54–68

[18] Chen, J., Wang, D. and Zhao, W. (2013) A task scheduling algorithm for hadoop platform. **Journal of computers**, 8 (4): 929–936

[19] Chen, Y. and Bordbar, B. (2016) "Dress: A rule engine on spark for event stream processing." In **Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies**. ACM. pp. 46–51

[20] Chen, Y. and Tino, P. (2018) Formalisation and verification of distributed rule-based expert systems. Paper submitted to journal Expert Systems with Applications.

[21] Cherniack, M., Balakrishnan, H., Balazinska, M. et al. (2003) "Scalable distributed stream processing." In **CIDR**. vol. 3, pp. 257–268

[22] Chomicki, J., Lobo, J. and Naqvi, S. (2003) Conflict resolution using logic programming. **IEEE Transactions on Knowledge and Data Engineering**, 15 (1): 244–249

[23] Cronk, R.N., Callahan, P.H. and Bernstein, L. (1988) Rule-based expert systems for network management and operations: an introduction. **IEEE Network**, 2 (5): 7–21

[24] Danelutto, M., Kilpatrick, P., Mencagli, G. and Torquati, M. (2017) State access patterns in stream parallel computations. **The International Journal of High Performance Computing Applications**, p. 1094342017694134

[25] Davey, B.A. and Priestley, H.A. (2002) **Introduction to lattices and order**. Cambridge university press.

[26] Dean, J. and Ghemawat, S. (2008) Mapreduce: simplified data processing on large clusters. **Communications of the ACM**, 51 (1): 107–113

[27] Dunkel, J., Fernández, A., Ortiz, R. and Ossowski, S. (2011) Event-driven architecture for decision support in traffic management systems. **Expert Systems with Applications**, 38 (6): 6530–6539

[28] Fernandez, R.C., Migliavacca, M., Kalyvianaki, E. and Pietzuch, P. (2014) "Making state explicit for imperative big data processing." In **2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)**. pp. 49–60

[29] FORGY, C. (1979) On the efficient implementation of production systems. **Ph. D. Thesis, Carnegie-Mellon University**.

[30] Forgy, C.L. (1981) Ops5 user's manual. Tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

[31] Forgy, C.L. (1988) "Rete: A fast algorithm for the many pattern/many object pattern match problem." In **Readings in Artificial Intelligence and Databases**. Elsevier. pp. 547–559

[32] France, R.B., Ghosh, S., Dinh-Trong, T. and Solberg, A. (2006) Model-driven development using uml 2.0: promises and pitfalls. **Computer**, 39 (2): 59–66

[33] GC, P.S., Sun, C., Zhang, H. et al. (2015) "Why big data industrial systems need rules and what we can do about it." In **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data**. ACM. pp. 265–276

[34] Giarratano, J.C. and Riley, G. (1989) **Expert systems: principles and programming**. Brooks/Cole Publishing Co.

[35] Gonzalez-Perez, C. and Henderson-Sellers, B. (2008) **Metamodelling for software engineering**. Wiley Publishing.

[36] Guerraoui, R. and Schiper, A. (1997) Software-based replication for fault tolerance. **Computer**, 30 (4): 68–74

[37] Guo, Y., Pan, Z. and Heflin, J. (2005) Lubm: A benchmark for owl knowledge base systems. **Web Semantics: Science, Services and Agents on the World Wide Web**, 3 (2-3): 158–182

[38] Gupta, A., Forgy, C. and Newell, A. (1989) High-speed implementations of rule-based systems. **ACM Transactions on Computer Systems (TOCS)**, 7 (2): 119–146

[39] Hammoud, M. and Sakr, M.F. (2011) "Locality-aware reduce task scheduling for mapreduce." In **Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on**. IEEE. pp. 570–576

[40] Hayes-Roth, F. (1985) Rule-based systems. **Communications of the ACM**, 28 (9): 921–932

[41] Hill, E.F. (2003) **Jess in action: Java rule-based systems**. Manning Publications Co.

[42] Hindman, B., Konwinski, A., Zaharia, M. et al. (2011) "Mesos: A platform for fine-grained resource sharing in the data center." In **NSDI**. vol. 11, pp. 22–22

[43] Hopgood, A.A. (2016) **Intelligent systems for engineers and scientists**. CRC press.

[44] Hwang, J.H., Balazinska, M., Rasin, A. et al. (2005) "High-availability algorithms for distributed stream processing." In **Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on**. IEEE. pp. 779–790

[45] Ibrahim, S., Jin, H., Lu, L. et al. (2010) "Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud." In **Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on**. IEEE. pp. 17–24

[46] Jin, J., Luo, J., Song, A. et al. (2011) "Bar: An efficient data locality driven task scheduling algorithm for cloud computing." In **Cluster, Cloud and Grid Comput-**

ing (CCGrid), 2011 11th IEEE/ACM International Symposium on. IEEE. pp. 295–304

[47] Jouault, F., Allilaire, F., Bézivin, J. and Kurtev, I. (2008) Atl: A model transformation tool. **Science of computer programming**, 72 (1-2): 31–39

[48] Kawakami, T., Yoshihisa, T., Fujita, N. and Tsukamoto, M. (2013) "A rule-based home energy management system using the rete algorithm." In **Consumer Electronics (GCCE), 2013 IEEE 2nd Global Conference on**. IEEE. pp. 162–163

[49] Kiran, M.S., Rajalakshmi, P., Bharadwaj, K. and Acharyya, A. (2014) "Adaptive rule engine based iot enabled remote health care data acquisition and smart transmission system." In **Internet of Things (WF-IoT), 2014 IEEE World Forum on**. IEEE. pp. 253–258

[50] Kolovos, D.S., Paige, R.F. and Polack, F.A. (2008) "The epsilon transformation language." In **International Conference on Theory and Practice of Model Transformations**. Springer. pp. 46–60

[51] Komazec, S. and Cerri, D. (2011) Towards efficient schema-enhanced pattern matching over rdf data streams. **10th ISWC. Springer, Bonn, Germany**.

[52] Koo, R. and Toueg, S. (1987) Checkpointing and rollback-recovery for distributed systems. **IEEE Transactions on software Engineering**, 13 (1): 23–31

[53] Kreps, J., Narkhede, N., Rao, J. et al. (2011) "Kafka: A distributed messaging system for log processing." In **Proceedings of the NetDB**. pp. 1–7

[54] Liang, S., Fodor, P., Wan, H. and Kifer, M. (2009) "Openrulebench: an analysis of the performance of rule engines." In **Proceedings of the 18th international conference on World wide web**. ACM. pp. 601–610

[55] Ligeza, A. (2006) **Logical foundations for rule-based systems**, vol. 11. Springer.

[56] Lucas, P. and Van Der Gaag, L. (1991) **Principles of expert systems**. Addison-Wesley Wokingham.

[57] McDermott, J. and Forgy, C. (1978) "Production system conflict resolution strategies." In **Pattern-directed inference systems**. Elsevier. pp. 177–199

[58] Mellor, S.J., Scott, K., Uhl, A. and Weise, D. (2004) **MDA distilled: principles of model-driven architecture**. Addison-Wesley Professional.

[59] Miranker, D.P. (2014) **TREAT: A new and efficient match algorithm for AI production system**. Morgan Kaufmann.

[60] Murata, T. (1989) Petri nets: Properties, analysis and applications. **Proceedings of the IEEE**, 77 (4): 541–580

[61] Nagl, C., Rosenberg, F. and Dustdar, S. (2006) "Vidre–a distributed service-oriented business rule engine based on ruleml." In **Enterprise Distributed Object Computing Conference, 2006. EDOC'06. 10th IEEE International**. IEEE. pp. 35–44

[62] Peters, M., Brink, C., Sachweh, S. and Zündorf, A. (2013) "Rule-based reasoning on massively parallel hardware." In **SSWS@ ISWC**. pp. 33–49

[63] Peters, M., Brink, C., Sachweh, S. and Zündorf, A. (2014) "Scaling parallel rule-based reasoning." In **European Semantic Web Conference**. Springer. pp. 270–285

[64] Peterson, J.L. (1977) Petri nets. **ACM Computing Surveys (CSUR)**, 9 (3): 223–252

[65] Reisig, W. (2012) **Petri nets: an introduction**, vol. 4. Springer Science & Business Media.

[66] Rosenberg, F. and Dustdar, S. (2005) "Towards a distributed service-oriented business rules system." In **Web Services, 2005. ECOWS 2005. Third IEEE European Conference on**. IEEE. pp. 11–pp

[67] Salatino, M., De Maio, M. and Aliverti, E. (2016) **Mastering JBoss Drools 6**. Packt Publishing Ltd.

[68] Shah, M.A., Hellerstein, J.M. and Brewer, E. (2004) "Highly available, fault-tolerant, parallel dataflows." In **Proceedings of the 2004 ACM SIGMOD international conference on Management of data**. ACM. pp. 827–838

[69] Sharma, T., Tiwari, N. and Kelkar, D. (2012) Study of difference between forward and backward reasoning. **International Journal of Emerging Technology and Advanced Engineering**, 2 (10): 271–273

[70] Shvachko, K., Kuang, H., Radia, S. and Chansler, R. (2010) "The hadoop distributed file system." In **Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on**. Ieee. pp. 1–10

[71] Sowa, J.F. et al. (2000) **Knowledge representation: logical, philosophical, and computational foundations**, vol. 13. Brooks/Cole Pacific Grove, CA.

[72] Stephen, J.J., Gmach, D., Block, R. et al. (2015) "Distributed real-time event analysis." In **2015 IEEE International Conference on Autonomic Computing**. IEEE. pp. 11–20

[73] Stolfo, S.J., Wolfson, O., Chan, P.K. et al. (1991) Parulel: Parallel rule processing using meta-rules for redaction. **Journal of Parallel and Distributed Computing**, 13 (4): 366–382

[74] Teymourian, K. and Paschke, A. (2009) "Semantic rule-based complex event processing." In **International Workshop on Rules and Rule Markup Languages for the Semantic Web**. Springer. pp. 82–92

[75] Toshniwal, A., Taneja, S., Shukla, A. et al. (2014) "Storm @ twitter." In **Proceedings of the 2014 ACM SIGMOD international conference on Management of data**. ACM. pp. 147–156

[76] Uml, O. (2004) 2.0 superstructure specification. **OMG, Needham**.

[77] Van Melle, W. (1980) A domain-independent system that aids in constructing knowledge-based consultation programs. Tech. rep., STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.

[78] Vavilapalli, V.K., Murthy, A.C., Douglas, C. et al. (2013) "Apache hadoop yarn: Yet another resource negotiator." In **Proceedings of the 4th annual Symposium on Cloud Computing**. ACM. p. 5

[79] Vijayaraghavan, A. and Dornfeld, D. (2010) Automated energy monitoring of machine tools. **CIRP annals**, 59 (1): 21–24

[80] Völter, M., Stahl, T., Bettin, J. et al. (2013) **Model-driven software development: technology, engineering, management**. John Wiley & Sons.

[81] Walzer, K., Groch, M. and Breddin, T. (2008) "Time to the rescue-supporting temporal reasoning in the rete algorithm for complex event processing." In **International conference on Database and Expert Systems Applications**. Springer. pp. 635–642

[82] Walzer, K., Schill, A. and Löser, A. (2007) "Temporal constraints for rule-based event processing." In **Proceedings of the ACM first Ph. D. workshop in CIKM**. ACM. pp. 93–100

[83] Wang, J., Deng, Y. and Xu, G. (2000) Reachability analysis of real-time systems using time petri nets. **IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)**, 30 (5): 725–736

[84] Warneke, D. and Kao, O. (2011) Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. **IEEE transactions on parallel and distributed systems**, 22 (6): 985–997

[85] Wehrmeister, M.A., Freitas, E.P., Pereira, C.E. and Rammig, F. (2008) "Genertica: A tool for code generation and aspects weaving." In **Object Oriented Real-Time**

Distributed Computing (ISORC), 2008 11th IEEE International Symposium on. IEEE. pp. 234–238

[86] Wickham, H. et al. (2011) The split-apply-combine strategy for data analysis. **Journal of Statistical Software**, 40 (1): 1–29

[87] Wu, S., Kumar, V., Wu, K.L. and Ooi, B.C. (2012) "Parallelizing stateful operators in a distributed stream processing system: how, should you and how much?" In **Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems**. ACM. pp. 278–289

[88] Zaharia, M., Chowdhury, M., Das, T. et al. (2012) "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." In **Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation**. USENIX Association. pp. 2–2

[89] Zaharia, M., Das, T., Li, H. et al. (2013) "Discretized streams: Fault-tolerant streaming computation at scale." In **Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles**. ACM. pp. 423–438

[90] Zaharia, M., Xin, R.S., Wendell, P. et al. (2016) Apache spark: a unified engine for big data processing. **Communications of the ACM**, 59 (11): 56–65

[91] Zhang, J., Yang, J. and Li, J. (2017) "When rule engine meets big data: Design and implementation of a distributed rule engine using spark." In **Big Data Computing Service and Applications (BigDataService), 2017 IEEE Third International Conference on**. IEEE. pp. 41–49

[92] Zhou, R., Wang, G., Wang, J. and Li, J. (2014) Runes ii: A distributed rule engine based on rete network in cloud computing. **International Journal of Grid and Distributed Computing**, 7 (6): 91–110

[93] Zhu, S., Huang, H. and Zhang, L. (2016) “A distributed architecture for rule engine to deal with big data.” In **2016 18th International Conference on Advanced Communication Technology (ICACT)**. IEEE. pp. 602–606