universität wien

MASTERARBEIT

Titel der Masterarbeit

„Workflow Watson (W2) -
A RETE-based Process Execution Synchronization Web Service"

verfasst von

Conrad Indiono BSc

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2013

Studienkennzahl lt. Studienblatt:	A 066 926
Studienrichtung lt. Studienblatt:	Masterstudium Wirtschaftsinformatik
Betreut von:	Univ.-Prof. Dipl.-Math. Dr. Stefanie Rinderle-Ma

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Wien, 30. Januar 2013

Unterschrift:

_____

Conrad INDIONO

Acknowledgements

The work presented in this thesis has been created in the context and as a preparation of C3Pro: project I743 funded by the Austrian Science Fund (FWF) and by the Deutsche Forschungsgemeinschaft (DFG).

I'd like to thank my adviser, Stefanie Rinderle-Ma, for the patience shown throughout the development of this thesis, and giving me a cozy working environment within the Workflow Systems and Technology Group to pursue my work.

Thanks to Jürgen Mangler for the pleasurable teamwork, senior advice, patience, technical support, and encouragements; Raffael Hickisch for giving me the motivation to finish the final stretch.

Thanks to my family, Mom and Dad for giving me a supporting environment at home; my brothers Rino, Ossa, Akira and AJ for getting my mind off work; and last but not least my wife, Uliana, for the heartwarming support and sleepless nights accompanying me throughout, even though we're physically separated more than 10600km from each other and living 6-7 hours apart. With this work done, we can finally live together — hopefully in the same time zone and without any long distances between us.

Conrad Indiono, January 2013

Contents

Chapter 1

Introduction

In todays complex and dynamic world, businesses need any edge they can to stay competitive. The *iron law of oligarchy* [1] tells us that it is up to the upper echelons within organizations to facilitate knowledge sharing and documentation to further their cause. From the inner perspective, organizations can follow best practices and guidelines to stay nimble and alert, keeping the inner infrastructure running smoothly. From the outside world, various effects keep companies on their toes. Market forces determine direction and affect what is done within a company. As such *time-to-market* is a major signaling metric to distribute deliverables at the correct time with the correct methodology. Regulations force organizations to change and stay flexible as well.

Processes inside companies count as assets. BPM [2] realizes the idea to document and model these processes. The next step is the computer-supported execution of documented processes. In some cases, activities are executed automatically without human intervention — as is the ideal of Service-oriented Architecture (SOA) [3]. In more realistic cases hybrid execution of processes combines human labor with automated entities (services); Process-Aware Information Systems (PAIS) [4] should be able to determine at which exact time an action has to be started and ideally completed. The goal is to eliminate as much waste as possible by staying lean [5]. The focus of this thesis is the computer-supported execution of processes. The idea is to augment arbitrary imperative workflow engines with a rule engine which, based on an event stream, can analyze and influence the execution of processes. This rule engine, named Workflow Watson (W2), can be used to solve various problems like inter-process synchronization and the enactment of security and compliance rules. Before we dive further into the details of W2, we first outline the field of process execution to contextualize the work in this thesis. Process Modeling is at the core of Business Process Management (BPM) [2], as the iterated creation and modification of process models is central to the business and process life cycle. Modeling processes can be achieved through various technologies, most of them classified as either imperative or declarative on one dimension and graphical or textual on the other. BPMN [6] is an imperative and graphical notation to represent process models which has gained wide acceptance. From the mathematical world petri nets [4] are another means to model processes, also in an imperative and graphical fashion. Declare [7] is a graphical, declarative language allowing modeling with constraints as the central focus point to define processes. The idea here being, that through the specification of a set of constraints the modeler can be absolved of the responsibility of specifying tricky activities that in combination satisfy certain constraints. The Declare engine takes the task of generating compatible models that follow the specified constraints. Some approaches combine ideas by allowing a graphical, imperative model be transformed into a declarative, textual

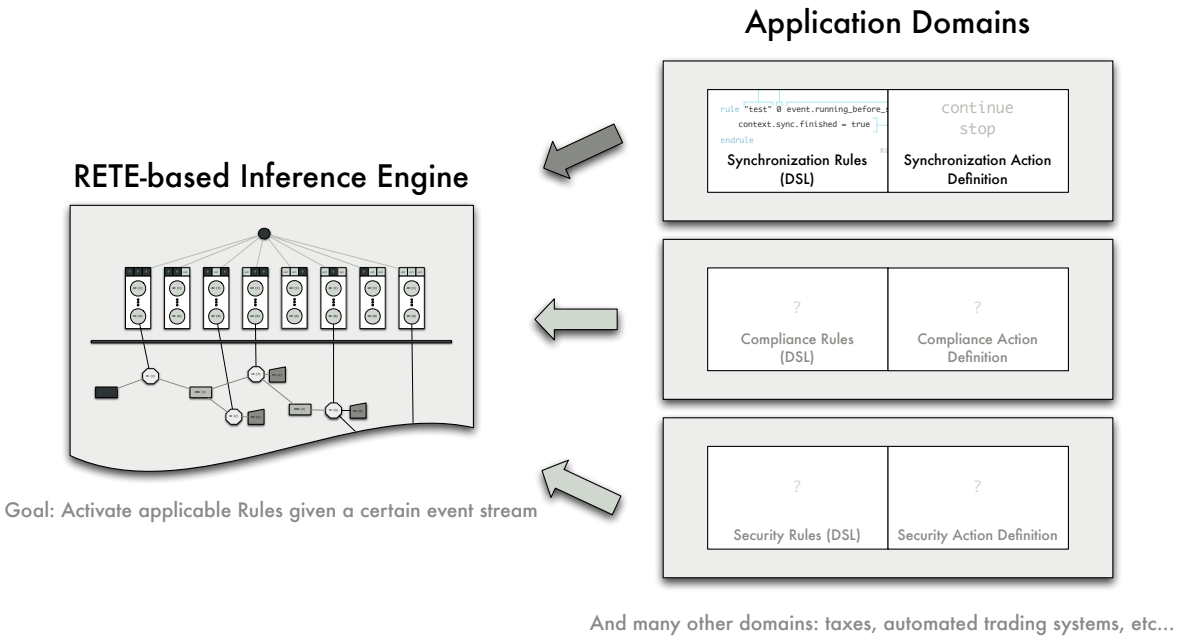

Figure 1.1: W2 Model

form (e.g. as ECA Rules) [8] which are then enacted by an engine.

Beside process modeling there are other issues — handled at run-time as well as design-time — that are topics of research relevant for this thesis. We dive into some of these topics in the next sections: synchronization, compliance checking, security rules and declarative process modeling. For these domains our W2 viewpoint of the world can offer solutions, and for one — synchronization — we concretely implement a solution based on the previous work of Mangler et. al [9]. What is this W2's viewpoint of the world? W2 combines a reasoning engine and DSLs. The reasoning engine is at the center driving the solutions in combination with custom grammars (DSLs) tailored for each application domain (Fig. 1.1).

1.1 Related Work

1.1.1 Synchronization

Mangler et. al [9] offer a solution for synchronization problems by introducing a rule-based synchronization engine. The problem of synchronization is defined as coordinating several activities within a single process instance, as well as various parallel running process instances. The solutions are based on various workflow patterns defined in [10]. Instead of either (1) integrating the workflow patterns inside the process model or (2) hard-coding them inside the process engine, the approach taken by [9] is through implementing a synchronization engine as an external subscriber to a process engine (CPEE). This rule-based approach allows the decluttering of synchronization primitives inside process models. These primitives include mutexes, locks, and triggers [9]. Further a resource-centric viewpoint for synchronization purposes turns out to be a natural way to think about resource control. All in all, the goal is to keep process models as simple as possible.

CPEE [11] is a cloud-based process execution engine that emits events at key stages of running process instances. These events are streamed to various subscribers that are

registered as external listeners. Both the subscribers as well as the CPEE engine itself follow the RIDDL specification [12, 13], allowing the description of both components as well as the composition of these as RESTful services. Incoming events can be handled by the subscribers as they see fit, and the voting capabilities of RIDDL are utilized to control the CPEE.

The assumption made by REMAR [9] is that a stream of specific events is the only input required to affect the running process engine. REMAR listens to specific events — *sync_before* and *sync_after* — that are emitted in between the execution of process activities by CPEE. The process engine models an extended process activity life cycle, that adds these two specific states during the execution of an activity for synchronization purposes. In [9] the default activity states among existing process engines are depicted as the following states: *calling*, *failed*, and *done*. In addition to these standard states CPEE adds two which an activity can hold: *manipulating* and *syncing*. The former is entered when the results of external service calls are being processed, and the latter — modeled by *sync_before* and *sync_after* — depict the various synchronization states the activity is currently inside. These two states are entered either before or after the activity state *calling*. Both help in the shaping of a running process instance through an external synchronization engine. This shaping occurs by utilizing the voting mechanism provided by RIDDL: as each process instance enters the *syncing* state whilst an active synchronization engine is asked to control the execution, the process instance awaits for a positive vote to continue execution — aptly called *continue*. Synchronization is achieved by suspending the execution of process instances until the rule-based synchronization engine tells the instances to continue.

1.1.2 Compliance Checking

Negative behavior of various major companies in the past — Enron and WorldCom to name a few — forced the development of various regulations and guidelines that future companies should and have to follow within their every day business processes. Staying compliant with these external regulations pose a challenge for today's organizations as the cost and effort of staying compliant is a huge burden [14]. [15] lists major problems concerning compliance. One of the major critical regulations are in the financial industry concerning money laundering. [16] confirms SOX being one of the most driving regulations steering towards compliance, as well as noting the high cost towards achieving compliance.

Various technologies and approaches exist. [17] defines a Formal Contract Language (FCL) to specify control objects and the associated task in a If-Then Rule format. These FCL specifications are connected with the activities within a process model to trigger compliance checks and alarm violations. [18] lists workflow time patterns similar to workflow patterns [10]. These can be codified inside compliance rules in order to handle those that dictate the time aspect within business processes.

A W2-based compliance checking algorithms can build upon the assumptions made in the synchronization section before. This is accomplished by listening on events emitted by the process engine and using them — including associated and relevant event attributes — as the input for compliance checking. Existing LTL-based structural compliance checkers could be utilized as black boxes to pattern match process structure. Some of these run at design-time [19, 20], and some at run-time [21]. The existing semantics of *skip* and *continue* from the synchronization domain can be imitated to signal compliance. A *stop* vote with relevant meta data can identify compliance violations and explain their causes. With our RETE-based approach we can identify the cause by traversing the tokens inside the Beta Network for bound fact values. Both design-time and run-time compliance checks are supported, as

the CPEE exposes topics and events that apply to both time dimensions.[22] touches the need of rechecking of compliance whenever new rules are added or existing rules changed. Through the *rule_change* event, which is emitted every time an existing rule in the repository is changed, we can perform the rechecks at the correct time.

[16] classifies the types of various compliance checking approaches. According to that classification our approach of using W2 as a compliance engine would constitute:

- Design-time as well as Run-time — Coupled to the CPEE, compliance checking can occur at design-time as well as run-time. The former is achieved through listening to events emitted during changes to the process model definition. Run-time compliance checking can be achieved by listening to events that are emitted before or after activity calls. Note that such events already exist as described in the synchronization section (1.1.1): *sync_before* and *sync_after*. Both of these hooking points to the running process instance serve as an opportune moment for compliance checking.

- *Forward* — *Forward* compliance checking is defined as checking for compliance while the process instance is running. In contrast *Backward* compliance checking is the method of checking for compliance after the fact. This can be achieved by means of process mining: through analyzing process logs. Naturally, only with the *Forward* method of compliance checking can the currently running process instance be affected. Our approach is strictly *Forward* compliance checking. ProM is one of the established tools for performing process log analysis [23]. [24] shows a method how, with the help of ProM, a *Backward* compliance checking approach can affect the currently running process instance, by feeding in partial logs to ProM and checking for compliance up to that point. [25] defines a query language that allows after the fact analysis. [17] calls *Backward* and *Forward* compliance checking as *retrospective reporting* and *automated detection* respectively.

- *Active/Passive* — *Active* compliance checking is defined as the compliance checking process controlling the execution of the running business process instance. And in contrast with *Passive* compliance checking the *business process execution components* control compliance checking operations. Our approach cannot be defined clear-cut on this dimension as both characteristics come into play. First, once the event that starts the compliance check service is emitted, the compliance checking facilities show an *Active* role until it casts its vote of *continue* for proceeding the business process. Once the vote is cast the CPEE takes over and shows a distinct *Passive* compliance checking role.

- *Task Checking/Process Checking* — With this classification a distinction is made on which level the compliance checking is performed. While *Task Checking* performs compliance checking on the activity level, *Process Checking* uses the whole process model as the subject for compliance checking. The different granularity levels are important, as the latter requires access to the whole process model. As defined by the *design-time* nature of compliance checking the process model changes can be the focus point for transferring the process model definition to the compliance engine. Once stored and memoized, access to the global process model can be performed at every compliance check. This shows that both *Task* as well as *Process Checking* is supported in W2.

- *Engine/Querying* — Our approach implements a dedicated compliance checking engine that implements the RIDDL specification and is hooked to the CPEE as an external

subscriber. For this reason, it is classified as *Engine*.

1.1.3 Security Rules

[26] defines a taxonomy for security rules. The authors also identify challenges and extract requirements that should be met before considering the use of security rules within an organization. The first challenge is the knowledge of the two different modeling approaches possible for security rules: (1) declarative and (2) imperative. Although the former allows high flexibility, that advantage comes with the high cost of impact analysis regarding potential security flaws. The goal is to have the process model become a reduced set of tasks, facilitating evolution. Ideally, security rules should be separated from the process model, with consistency checks made possible through a central repository. Another challenge: separating security rules from other constraints. This requirement is motivated by the fact that verification effort can be reduced by keeping each rule base separate and small. The next challenge states the need to map rules to process activities. This is naturally required once the previous requirement of keeping the security rules from process models separate is upheld. This mechanism needs to be unambiguous and efficient. The next two challenges concern evolution for (1) process models and (2) security rules. The first states that whenever a process changes, the corresponding security rules should be validated at both build and run time. Security rules evolution states that the repository should allow the administration of security rules: by adding, updating and deleting them.

Following these challenges, we can conclude that W2 meets most of them to serve as the basis for a security rule enforcement engine. It acts as a central repository for security rules with the ability to administer them: adding, deleting and editing. The separation of security rules from the process model means that the model is free from distracting security logic. W2 rules are inherently declarative, and as such the high cost of impact analysis applies. A concrete mapping function between security rules and process activities needs to be defined concretely. Even though a RETE-based inference engine would maximize separation between general constraints and security rules. An explicit separation by maintaining two rule bases would reduce verification effort considerably. W2 separates the DSL from the RETE-based inference engine and by adapting the DSL grammar we could have a security-specific DSL for security experts.

1.1.4 Declarative Approach to Process Modeling

DECLARE [7] is one of the declarative process modeling approaches. Its motivation is based on the disadvantages underlying imperative process modeling approaches: inflexibility. Once a step-by-step structured process model has been defined and all decisions made at design-time, users have little to no influence on the process once executed. ADEPT [27] solves the same inflexibility problem by defining a set of change operations to *ensure correctness and consistency*. These operations can be applied to running process instances. [7] criticizes ADEPT of users being required to have modeling knowledge for applying these operations. DECLARE additionally solves the issue of keeping activities compliant to some defined constraints. This is one of the areas where the declarative approach excels at. It is possible to automatically generate compliant process models, by being constraint-centric during process modeling.

[28] is considered the first attempt of applying ECA rules to Workflow Management Systems (WFMS). [8] outlines a possible way to implement process execution based on transforming a graphical — and imperative — process model into a compiled hierarchical tree

representation of ECA rules. This allows process modelers to procedurally define their models, and benefit from the underlying declarative system. The lack of visualization is mentioned as the reason for the inappropriateness of declarative process modeling, as the rules are hard to understand without visual support. The implementation is based on an *active database*, with triggers emitting events, and rules modeled using stored procedures. The action part consists of arbitrary SQL statements — even potentially unsafe ones.

Compared to the DECLARE approach, ours — coupled with a process execution engine (CPEE) — allows procedural modeling of process models. [8] argues that this is the natural way how humans think, and the difficulty of following non-graphical rules — note that DECLARE is defined as a graphical modeling language. The advantage of the DECLARE approach could be emulated by *outsourcing* the compliance checking mechanism to an adapted W2 engine, which listens to a process model being constructed at *design-time*. The other advantage of automatically generating compliant process models requires an additional subscriber to the CPEE that is specialized in the generation with collaboration of a W2-based compliance engine. This idea neatly follows the concept of *separation of concerns*. Each component is responsible for their special domains: CPEE for process execution, a W2-based compliance engine for verification, and the just described process model generator for doing just that. Compared to [8] our approach distances itself from being database-centric and being more SOA-compatible to enable cooperation with related web services. We also provide a safe sandbox environment for the action part to avoid running potentially insecure code. Additionally, we do not base the execution of process instances themselves to be based on a rule engine.

1.2 Workflow and Rule Engine Interaction

As stated above, for this thesis we used the cloud process execution engine (CPEE). Events emitted from the CPEE are grouped according to several topics. These topics include the currently running process instance's activity state changes, process description changes, position changes (of the currently focused activity), changes to the process model itself and various other observable attributes of the process engine. By making these aspects of the engine transparent as subscribable event streams, connected listeners can handle various topics relevant during process execution. These topic-grouped events exist for the external services as well, as long as they follow the RIDDL specification. This fact allows the event subscription of other subscribers on the rule-based synchronization engine itself, meaning that state changes and internal context changes of the synchronization engine are transparent to external subscribers as well. This transparency and unlimited listening of event streams can lead to a *behavioral shaping network* (see Fig.1.2) of subscribers that each handle specific aspects of the process engine or synchronization engine, allowing separation of concerns and modularity of services. Each element of the network can shape each other in a dynamic fashion.

The rule-based synchronization algorithms implemented with REMAR [9] use a naive *unify* pattern matching algorithm in order to process matching rules from incoming events. The goal of this thesis is to implement the same interfaces as REMAR but improving the inner pattern matching algorithm with RETE. Through decoupling the rule engine from the DSL we can adapt W2 to other domains. Some of these application domains have been discussed in the related work (see Sec. 1.1).

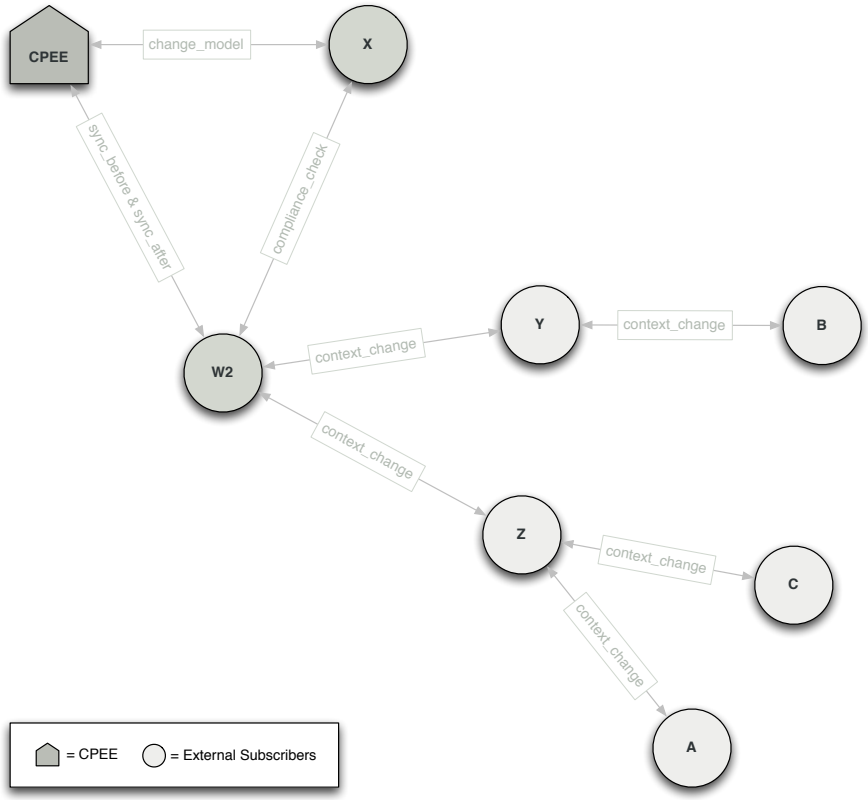

Figure 1.2: Behavioral Shaping Network

1.3 Research Methodology

In order to realize this thesis we used a simple research methodology: based on an extensive literature review covering workflow execution, rule engines and more generic first-order logic and the RETE algorithm, we created a prototypical implementation, which was then benchmarked under various scenarios.

1.4 Contribution

The contribution of this thesis is two-folded:

1. The development of W2, which improves upon REMAR by improving the core rule matching algorithm. Beside the inference engine we define a DSL layer that is adaptable to different application domains by modifying its grammar (see chapter 3). W2 works in cooperation with the CPEE, allowing a hybrid approach. Combining the imperative workflow engine with a declarative rule engine allows us to exploit the power of rule-based systems with the understandability of traditional workflow systems. Complementing processes with rules and facts allows the shaping of the execution of process instances.

2. Domain-specific optimizations which we can apply on top of the RETE algorithm. Conjunct ordering [29, p.284] is NP-hard. [30] shows that there are general heuristics one can follow in order to find the optimal ordering of conditions within rule definitions. We em-

pirically show that specific ordering of conditions are responsible for consistent speedups across many dimensions (see chapter 4).

1.5 Summary

In this chapter we defined the context of W2 as a synchronization engine that is designed as an external subscriber to listen to the event stream of a process engine (CPEE).

Beside synchronization there are other uses of rules: we have explained how rules can be utilized in the context of process execution. There are several application domains (compliance checking, security policies etc.) that could use rules for problem solving. W2's model can be used as the basis for a solution in those domains by reusing the RETE inference engine and defining the domain-specific DSLs (e.g. compliance checking DSL or security policies DSL).

W2's goal is to strictly emulate the behavior of REMAR, following the interfaces outlined by the latter, although with an improved pattern matching algorithm.

Chapter 2

Theoretical Background For Inference Engines

Knowledge-based systems are programs that exhibit human intelligence. They behave in a manner that we as humans consider natural. Rational humans not only possess knowledge about their surroundings — in form of facts that they consider to be true — they also follow a reasoning process to derive new facts from already previously known ones.

Knowledge-based systems show desirable characteristics that are useful. For one, they have the ability to explain — through the actual facts and the reasoning machinery — as to how a certain conclusion was met. Secondly, knowledge-based systems are structured in a modular fashion allowing new facts to be added incrementally which may depend on already asserted facts.

2.1 Expert Systems

Both of these characteristics are especially important for a certain class of knowledge-based systems: expert systems (ES). ESs are typically used as interactive consultation programs designed to aid humans in decision making. Users of expert systems consult the program by answering relevant questions for the specific domain the system was constructed for. These answers are the facts that the system uses to reach a conclusion. They are also used as jumping points that lead in to the next logical questions. The system offers its advice based on the facts supplied by the user, and along with it an explanation on how the conclusion was reached. This ability to explain the reasoning process is important, as faulty conclusions can be identified and corrected on the spot. Domain experts are used to teach the system, whose job would be unnecessarily complex without a way to identify the believes of the system. [31]

The goal of an expert system is to emulate the level of competence of a similar expert would have in some domain. In order to achieve that, the system requires the same knowledge and believes that distinguishes a novice from the expert. The five stages of the Dreyfus model classify the different human skill levels ranging from novice to expert [32]. The main distinction between the first level (novice) and the fifth level (expert) are the approach to decision making: whereas the expert follows his or her intuition, the novice — actually, all levels below the expert level — takes a more rational approach for making decisions. The rational conclusions made are accomplished by mostly following rigid rules he learned, whereas the expert no longer relies on the same rules but has an intuitive grasp of the situation to guide his decisions. Thus, Hubert Dreyfus argues [33, p.8] that a knowledge-based system

approach is misguided, if the goal is to duplicate human-level intelligent behavior. The skill disparity among chess players — chess master vs chess novice — is an example of the different decision making approaches. Whereas the novice needs to figure out the situation by individually scanning through his learned rules, the master just intuitively *sees* the next move by honing in on the most important rules that apply in that specific context. Whether or not a knowledge-based system approach is misguided, its characteristics are still useful for real world problems. The explanatory facilities of the knowledge-based approach is certainly useful to determine how a conclusion has been made. The set of acquired facts that represents the knowledge of a domain expert is stored in a KB. The other critical component of a knowledge-based system is the inference engine (IE) that represents the reasoning process. Both of these components satisfy McCarthy's declarativism [31], which states that *"a program's knowledge about objects and relations should be explicitly stated, in order for other programs to reason about that knowledge"*.

2.1.1 Expert Systems throughout history

[31] gives a quick overview of the time line of expert systems. Throughout history, expert systems have been developed and employed in a variety of industries and domains. The first example is DENDRAL [34], considered as the progenitor of expert systems, which was devised as a research project at Stanford University. It led the way and legitimized the knowledge-based approach and was soon followed by MYCIN [35] — an interactive consultation program that aids physicists in prescribing medications against bacterial maladies. Another popular example is XCON [36] — developed by the Digital Equipment Company (DEC) — which performed its job *"300 times faster than human engineers"*[31] in configuring minicomputers. The efficiency of the system in juggling thousands of components and completing a full system build with exact precision helped DEC in saving millions of dollars every year.

Some expert systems were built to interpret signal data to form hypotheses. General Motors built the Charley Program that analyzes vibrations of mechanical equipment for troubleshooting. This system was popular in the automotive industry and other manufacturing equipment companies. Schlumberger's Dipmeter Adviser [37] works in a similar fashion but works with streams of data in order to find the optimal location for oil exploration. Both follow the common thread of externalizing the knowledge of senior specialists to *"capture and preserve rare corporate expertise"*[31].

Working with large amounts of data helped American Express [38] devise an expert system in assisting the work of manual authorization of customer charges. The system makes a recommendation, and along with it an explanation for the reasoning with the associated data to confirm the reasoning process. In this way millions of dollars in bad debt can avoided every year. Similarly, DEC developed Lend Lease [31] for a *"large construction firm to estimate the total completion time for high-rise buildings"*. These conclusions are made within +/- 10% accuracy after the first meeting with customers, allowing accurate project planning.

Expert systems are also deployed world-wide where complex systems are in place. This is the case where intricate tax laws are involved or bureaucratic regulations are to be followed. The US Internal Revenue Service has the Taxpayer's Assistant Expert system [31] *"answering taxpayers' queries in the form of accurate tax information"*, and the British social security administration have expert systems that *"assist clerks in answering queries from citizens regarding their pensions"*.

Recently, IBM developed Watson [39], a supercomputer that holds the full text of Wikipedia and several other taxonomies, ontologies such as DBpedia, WordNet and Yago. Its

goal was to have the ability to answer any open-ended question posed to the system without access to the Internet. Its ultimate test was its participation at the quiz show "Jeopardy" in the year 2011 competing against the then champion Ken Jennings — who had a 74 winning streak — and Brad Rutter [40]. Watson received the first prize decisively against its contenders and proved its reasoning abilities. One of the challenges lied in the fact that questions were posed in a natural language — English — requiring Watson to parse the questions using Natural Language Processing (NLP) techniques. The parsed concepts are then queried against its KB and pruned for relevance to those rules and facts that are specific to the question. Thus, Watson is not a pure knowledge-based system but also used machine learning techniques to reach a conclusion: the correct answer.

2.2 Key Concepts Underlying Knowledge-based Systems

Having outlined the advantages and the goals of a knowledge-based system, we will now identify the common elements underlying these systems. We have established the fact, that all knowledge-based systems are composed of two primary components: (1) the knowledge base (KB) to hold the facts and believes of the program and (2) the inference engine that transforms existing and explicitly stated facts to implicitly non-stated ones. [31]

2.2.1 Knowledge

What kind of knowledge and facts are stored inside the KB? [33] defines knowledge to be a *"relation between a knower [...] and a proposition"*: "Daniel knows that the sky is blue". In this example the knower — Daniel — considers the proposition "The sky is blue" to be true. The goal of the proposition is to classify the world into two categories: true or false. These judgments by the knower about his or her world, allows programs to model the world correctly. The proposition "The sky is blue" tells us that the world in which Daniel lives it is considered that another entity — the sky — is of the color blue, and not of any other color.

2.2.2 Representation

Having the knowledge is not enough. Getting these propositions into the KB requires some way of representing them symbolically. [33] considers representations to be a relationship between two domains, where one of them taking place of the other. Concretely, the symbols "13/8/2008", "13th August 2008" and "Second Wednesday of August in the year 2008" all represent the 13th day of the month of August in the year 2008. The symbols making up the English sentence "The sky is blue" represents the fact, that in the world in which the person stating the statement lives, it is considered to be true that the sky has in fact the color blue.

Knowledge Representation is thus *"the field of study concerned with using formal symbols to represent a collection of propositions by some putative agent"* [33].

2.2.3 Reasoning

While there can be possibly unlimited propositions for a given domain that a program could use inside its KB, it is not practical to pursue the possibly endless collection of facts. Through reasoning we can actually accomplish the job of deriving implicit propositions from already explicitly stated ones. It is in this context useful having the representation of propositions as manipulatable symbols [33], as they can be used to construct new ones, in effect constructing new propositions.

First put forward by Gottfried Leibniz in the 17th century [33], reasoning is not unlike arithmetic. Consider the operation "+" (addition) on the symbols "5" and "10". Applying said operation results in a new symbol "15". Similarly, through reasoning we can derive new propositions from existing ones. The proposition "Daniel likes cheese" and another "This year's trends show a global cheese shortage" results in a new proposition "This year's trends show global shortage of food Daniel likes". This type of reasoning is called logical inference, and is similar to arithmetic in that the initial propositions are calculated to form symbols that represent a new proposition.

Why is knowledge representation and reasoning useful? In order to answer this question we consider a program that internally has hard-coded "knowledge" and rules it is supposed to follow. Due to the rules being hard-coded and no reasoning process being followed the program might explain its actions by printing out its internal data structures. In contrast a chess program using a knowledge-based approach might explain its action of taking the queen in the center position with your own queen through the — albeit simple — propositions: "If the opposing queen endangers a high-value token and is unprotected, attack it with your own queen", "The king is a high-value token", "The enemy queen is unprotected", "The queen is able to attack the enemy queen". Together these propositions allow the explanation of the program's action, even though in reality such a move by the enemy player might constitute a mistake.

2.2.4 Logic

For reasoning to work properly it requires an underlying logical system. Consider a KB that holds two propositions or facts. Further consider the context to be an agent which takes actions according to internal as well as external states. The first proposition says

$$Hungry_{agent} \Rightarrow SeekFood_{agent} \tag{2.1}$$

and the second fact states that

$$Hungry_{agent} \tag{2.2}$$

meaning that the agent is in the internal state of being hungry. From these two propositions

$$SeekFood_{agent} \tag{2.3}$$

can be inferred given the two assertions to the KB. Thus the KB entails $SeekFood_{agent}$, stated as

$$KB \models SeekFood_{agent} \tag{2.4}$$

Informally — given $A \models B$ — logical entailment states that if A is true then B must also be true [29, p.201].

Logical entailment is employed in order to derive conclusions from existing facts. For this inference step to work it is imperative that a knowledge representation language has a well-defined notion of entailment. [29, p.203] likens the derivation of an entailed sentence to finding a needle in a haystack. In this metaphor the haystack represents all the consequences resulting from a KB, whereas the needle represents the single entailed sentence that we are looking for as an answer. The goal is then to find this needle in the haystack, and being able to explain the conclusions reached. This is depicted as

$$KB \models_i A \tag{2.5}$$

where we entail a sentence A from a KB using a specific sentence or proposition i.

At the end of the process of deriving sentences — with the help of inference rules [29, p.211] — we reach a proof [29, p.212]. These patterns of inference serve as an alternative to enumerating models using model checking [29, p.202]; models are sets of truth values for the symbols in propositional logic [29, p.201]. One inference pattern is *modus ponens* [29, p.211] formalized as

$$\frac{a \Rightarrow B, a}{B} \tag{2.6}$$

stating that given the rule $a \Rightarrow B$ and a we can conclude that B holds. Another pattern is the *And-Elimination* [29, p.211] formalized as

$$\frac{a \wedge B}{a} \tag{2.7}$$

saving us the evaluation of B given $a \wedge B$, or a whichever is more cost-effective to evaluate.

An inference algorithm has two important properties. One is soundness; an inference algorithm is sound, if it preserves truth by deriving only entailed sentences [29, p.203]. The other property is completeness; an inference algorithm is complete if it can derive any entailed sentences, which is problematic in cases the consequences of a KB is infinite [29, p.203].

Allen Newell states [33] that a knowledge-based system can be observed on two differing levels. One is the knowledge level, where we ask questions regarding the representation language, its semantics and its expressiveness. The other is the symbol level, where we concern ourselves with the computational aspects, including computational architecture, data structure, and reasoning procedures.

2.3 Propositional Logic

2.3.1 Syntax

```
Symbol = P | Q | R ...
AtomicSentence = True | False | Symbol
ComplexSentence = (see definition below)
Sentence = AtomicSentence | ComplexSentence
```

The syntax of propositional logic defines the valid grammar [29, p.205] of it as a representation language. At the lowest level we have symbols which are represented by uppercase letters. Symbols in parallel with the strings True or False form atomic sentences, which is one of the sentence types. The other are complex sentences which combine logical connectors with sentences. All the obvious logical connectors such as negation, conjunction, disjunction, implication and bi conditionals are supported for complex sentences in propositional logic. *ComplexSentence* can be one of $\{\, \neg Sentence$, $(Sentence \wedge Sentence)$, $(Sentence \vee Sentence)$, $(Sentence \Rightarrow Sentence)$, $(Sentence \Leftrightarrow Sentence)\,\}$.

2.3.2 Semantics

In propositional logic the KB is a conjunction of all asserted sentences. Semantics specify how to recursively *"compute the truth value of any sentence, given a model"* [29, p.206]. We start with atomic sentences where we can map the True and False strings as their respective true and false values in every model. From these atomic sentences we can fix the complex sentences using the already fixed values [29, p.206]. For complex sentences [29, p.206] states *"For any sentence s and any model m, the sentence $\neg s$ is true in m iff s is false in m"*.

The bottom up processing from atomic sentences to complex sentences *"reduces the truth value of a complex sentence to the truth of simpler sentences"* [29, p.206]. The end result is a truth table summarizing the truth values of all the sentences within the KB [29, p.206].

Logical inference with propositional logic — or propositional entailment — is co-NP-complete [29, p.210].

2.4 First-Order Logic

First-Order Logic is a knowledge representation language that extends the grammar of proposition logic with namely two features: variables and quantifiers. Using this formal language gives us the ability to formulate ideas expressed more intuitively compared to propositional logic in order to reason about, learn from, plan or explain statements. [29, p.242]

2.4.1 Programming Languages as Representation Languages

A short comparison between programming languages and a formal representation language highlights the features we expect from a representation language. First, we establish that programming languages perform ad hoc representation of facts via custom data structures [29, p.240]. Examples include representing a geographical map using a two-dimensional array, representing keywords in sparse-matrices for search engines, or a one-dimensional array representing a single classroom of students. Relational Database Management Systems (RDBMS) were conceived in order to manage — via *store* and *retrieve* — data in a domain-independent fashion [29, p.241]. What is expected of a representation language is to not only store and retrieve facts, but to also derive new facts from already existing ones in an domain-independent fashion [29, p.241].

With a programming language as a representation language all updates to the ad hoc representation — data structures — are procedures that are tied to the domain. The implementor's domain knowledge is implicitly embedded, in a sense that the domain knowledge is intertwined in an implementation-specific way. This makes general inferencing on that knowledge impossible. Unlike with the procedural approach using a general programming language, a declarative approach separates the inference mechanism from the domain knowledge component, allowing a domain-independent way to derive new facts. [29, p.241]

Another feature lacking from programming languages is composition. Using a representation language should allow the composition of complex sentences from simple ones. Propositional Logic and other representation languages achieve this by connecting sentences using logical connectors. The meaning of the sentence $S(1,2) \vee S(2,4)$ is inherently dependent on the meaning of its parts $S(1,2)$ and $S(2,4)$. [29, p.241]

2.4.2 Natural Languages as Representation Language

Comparing natural languages with formal representation languages show us further desirable features and the unsuitability of natural languages as representation languages.

First, with natural languages the context of the speaker is an important aspect that drives the meaning of his sentences. The sentence "Watch out!" can have different meanings depending on the context. In one, it might mean that an aggressive lion has broken from a zoo or it might mean an uncontrollable train is speeding down the rails, each worthy of our attention. This understanding of the importance of context leads to the question of *"how the context itself can be represented"*. [29, p.242]

Secondly, related to the first point, words themselves can have different meanings: ambiguity. The word "brush" can mean the act of brushing or the tool with which we brush something. Ambiguous words are made exact with the help of context, and through the use of preceding as well as succeeding words to disambiguate. [29, p.242]

Finally, as is the case with programming languages, sentences stated with natural languages are of non-compositional nature. Here again, the surrounding words tie the meaning of the actually stated sentence. [29, p.242]

[29, p.242] thus summarizes the characteristics of First-Order Logic (FOL) the following way:

1. FOL is a representation language that is — following its propositional logic roots — declarative.
2. FOL has *"compositional semantics"*, allowing mixing and matching of different statements using logical connectors.
3. Sentences in FOL are *"context-independent"* and *"unambiguous"*.
4. FOL extends propositional logic with *objects*, *relations* and *functions*. Every day human existence can be usefully thought of as dealing with these three concepts.

2.4.3 Ontological and Epistemological Commitment

Representation Languages make assumptions about various issues. The first one, ontological commitment, are the assumptions made about the world by a representation language. In propositional logic facts or sentences inside the world either *"hold or do not hold"*. Stated differently sentences are either true or false. First-Order Logic extends these assumptions made by propositional logic and states *"that the world consists of objects with certain relations among them that do or do not hold"*. FOL can be seen as a special purpose logic derived from propositional logic that treats objects and relations as first-class citizens within the logic. Having concepts as *"first-class status"* inside a logic means not representing those concepts inside the KB itself. A similar specialized logic is linear temporal logic which treats the time dimension as a *"first-class object"*. Its ontological commitment lies in treating facts as having a time aspect to them. Facts are held at particular times and can be ordered by this time property. Higher-Order Logic (HOL) considers relations and functions as objects. This assumption allows assertions to be made about all relations inside the KB, which is not possible in FOL. Higher-Order Logic is thus more expressive in this specific regard compared to FOL. [29, p.244]

Epistemological commitment is another aspect of representation languages. It documents the possible values or states each fact within the representation language holds. For propositional logic, FOL and temporal logic there are three possible states: true, false, or unknown.

Fuzzy logic specify the possible states as a *"degree of belief"*. Here the value 0 represents facts or sentences considered unlikely to be true, whereas a value of 1 represents facts that are considered to hold all the time. Values inside this range $[0, 1]$ show how strong a fact is considered to be true. [29, p.244]

2.4.4 Syntax

Symbols

The grammar of FOL is similar to propositional logic with additional rules for quantifiers and variables. On the atomic level we can identify three different symbol types: constant, predicate and function symbols. The first — constant symbols — depict objects, which can hold many different names. But the act of naming of objects is optional. Objects can be nameless and still exist. Relations are represented using predicate symbols, and the last concept — functions — are depicted using function symbols. For example, the predicate symbol $RightHand(x)$ with $x = John$ means that $RightHand(x)$ represents the actual right hand of "John". Notice that the hand is an unnamed object. x in contrast is an object that is named "John". The two last symbol types have arity, which states the number of input parameters the respective symbols accept. The *RightHand* predicate in the previous example has an arity of 1. A predicate symbol *Parent* would have arity 2, which requires two objects to designate the parent relationship between objects. Such a symbol would be used like $Parent(x, John)$ stating x being the parent of *John*. [29, p.246 - p.247]

The equality symbol serves two purposes. For one it is required to build up atomic sentences using terms. One example to satisfy the grammar rule

$$Term = Term \tag{2.8}$$

is $x = Anna$. The x in this example is a variable, and the second term *Anna* is a constant symbol stating that *Anna* is an object. The other purpose is for allowing several terms to refer the same object. In our example x and *Anna* are terms that refer to the actual person (object) named "Anna".

Interpretation [29, p.246] states that Interpretation *"specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols"*. Even though sentences can be syntactically correct, it does not mean that the semantic interpretation is the one an author intended. It is possible to say that the weather is both *Hot* and *Freezing* at the same time. Models with obvious paradoxes have to be ruled by the KB [29, p.247]. In FOL, using the $Term = Term$ rule from 2.8, we might state that

$$Hot = \neg Freezing \tag{2.9}$$

and

$$Freezing = \neg Hot \tag{2.10}$$

and thus making sure that if either one holds, the other will automatically not. Having such a statement makes the following statement in any one the models semantically impossible: $Hot \wedge Freezing$.

Term

A term, as described above in the symbols section 2.4.4, is *"a logical expression that refers to an object"*. Even though constant symbols are classified as terms as well, it might be impractical to name every object inside the universe of an application domain. These are objects that can and should remain nameless. Complex terms in the form of $Function(Term, ...)$ are just complex names and are not equal to routines in programming languages. The difference lies in the fact that with the latter we cannot reason about the terms. Even without defining what a *Finger* is or does, we can still state where those objects are attached, and that most humans have these objects. [29, p.248]

Atomic Sentences & Complex Sentences

We mentioned before that an atomic sentence can be constructed using the $Term = Term$ rule (2.8). Another rule

$$Predicate(Term, ...) \tag{2.11}$$

allows the construction of the same. Atomic sentences, generally, are composed of objects and relations that are referred by terms and predicate symbols respectively. They state facts:

$$Brother(John, Patrick) \tag{2.12}$$

$$Mother(Anne) \tag{2.13}$$

$$x = John \tag{2.14}$$

and are *"true in any given model, under a given interpretation, if the relation referred to by the predicate symbol holds among the objects referred by the arguments"*. Like in propositional logic it is possible to construct complex sentences by combining them using logical connectives. [29, p.248]

Quantifiers

Quantifiers are one of the new concepts added in FOL compared to the propositional logic grammar. They solve the problem of repeating sentences in some schema to a set of different variables. In a sense quantifiers can be equated to loops in programming languages, which allows iterating over a set of objects and perform some kind of manipulating operations over that set. The first quantifier is called the universal quantifier which says that *"for all"* specific objects that are mapped to a variable, a specific sentence holds. Quantifiers use lowercase variables, which are classified as terms too. Terms without variables are called ground terms. Variables and terms can serve as the input parameter of a function. [29, p.249]

A universally quantified sentence:

$$\forall x P \tag{2.15}$$

, where P is any logical expression, states that P is true for every object x. This example universally quantified sentence replaces a whole list of sentences that would be required to state the same effect.

Whereas universal quantifiers state something for **all** objects that match, existential quantification allows a statement be made for **any** variable that happens to match a specific sentence. [29, p.251]

The existentially quantified sentence

$$\exists x P \tag{2.16}$$

, where P is again any logical expression, states that there exists an x such that P is true. In other words, there is at least one object x where for which P is true [29, p.250]. Quantifiers express the properties of object collections, which replaces object enumeration by name — propositional logic is less expressive in this regard. Quantifiers can be nested: $\forall x[(Happy(x) \vee (\exists x Father(Anna, x))]$ [29, p.251] and are simply conjunctions (for the universal quantifier) and disjunctions (for the existential quantifier) over the object universe [29, p.252].

2.4.5 Using FOL

Having defined the syntax of FOL, we can now use the language to represent knowledge that is considered fact for a given part of the world: the domain. Sentences are asserted to the KB and are thus also known as assertions. Once enough knowledge has been captured into the KB interesting questions can be asked to the KB, which the systems gives answers to. The questions herein are called queries or goals, whereas the answers are in the form of a substitution or binding list. [29, p.253-p.254]

An example is in order. Given a query that asks the KB for a list of objects that are considered a *person*:

$$ASK(KB, \exists x Person(x)) \tag{2.17}$$

the resulting answer can be seen as a substitution list:

$$\{x/John, x/Anna\} \tag{2.18}$$

A substitution is represented as a tuple of two values separated by a forward slash (/) [29, p.254]. The left value states the variable that is the target of substitution and the right value is the substituted value. In this case the answer tells us that there are two objects — one *John*, the other *Anna* — that satisfy the condition of the object being a person. Without peeking into the KB we might assume that the assertions

$$Person(John) \tag{2.19}$$

and

$$Person(Anna) \tag{2.20}$$

has been submitted to the KB. For these sentences are the most logical assertions that could be made to declare these two objects as persons. But these are not the only candidates to declare objects as persons since there are possibly unlimited ways to declare such sentences. One of them might state persons to be living "things" that have specific physical features, such as having a head, two arms and two legs:

$$\forall x Living(x) \wedge Head(x) \wedge RightArm(x) \wedge LeftArm(x) \wedge RightLeg(x) \wedge LeftLeg(x) \Leftrightarrow Person(x) \tag{2.21}$$

Even though this statement is comical as the definition of a person using physical characteristics is not that simplistic, it shows that being able to state two unrelated sentences and combining them together using logical connectives is powerful. From that single assertion we can derive not only that *John* and *Anna* as persons, but also that they have the physical attributes of a person. The query

$$ASK(KB, \exists x Head(x)) \tag{2.22}$$

will then also return the substitution list

$$\{x/John, x/Anna\} \tag{2.23}$$

per definition of equation 2.21, which states that for an objects to be considered as a person, it also requires a head.

The preceding sentence is also called an axiom. Axioms are elementary sentences that state plain facts, such as $Male(Jim)$. These are the kind of sentences that are needed in all kind of domains, as they provide basic facts. From such sentences useful conclusions can be derived, called theorems. [29, p.255]

Theoretically a KB can be constructed strictly using axioms only, as all theorems are derived from the respective axioms. We can say that *"theorems do not increase the set of conclusions that follow from the knowledge base"* [29, p.255]. They do however, serve as computational shortcuts in that they relieve us from the need to work from *"first principles"* every time we use the reasoning facilities [29, p.255].

2.4.6 Inference Rules for Quantifiers

First-Order Inference (FOI) can be accomplished by utilizing model checking [29, p.202]. This can be accomplished by converting the KB in FOL into propositional logic, a process called propositionalization (described in 2.4.7). Beside this conversion, we still require the semantics to handle inference for the newly added functionalities: universal and existential quantifiers. These are called universal instantiation (UI) and existential instantiation (EI) respectively [29, p.273].

For both of these instantiations we define the following substitution function:

$$SUBST(\theta, a) \tag{2.24}$$

where θ is the set of variable substitutions (e.g. $\{v/John\}$) and a representing a sentence in FOL. $SUBST(\theta, a)$ then represents the sentence a with its variables set according to the variable mapping in θ. $SUBST(\{x/John\}, Bill = Father(x))$ would be transformed into the sentence $Bill = Father(John)$, with the meaning of the resulting sentence being: Object with the constant symbol *Bill* is the father of the object with the constant symbol *John*. [29, p.273]

Universal Instantiation (UI)

$$\frac{\forall v\ a}{SUBST(\{v/g\}, a)} \tag{2.25}$$

2.25 describes the rule of universal instantiation, which says we can infer sentences by substituting a *ground term* (g) for any variable (v) within the original sentence a [29, p.273]. Consider the sentence

$$\forall x Person(x) \vee CommittedMurder(x) \Rightarrow Evil(x) \tag{2.26}$$

implying that any person that commits a murder being evil. With the assertion of the sentence

$$CommittedMurder(John) \tag{2.27}$$

and the application of UI on the sentence 2.26 will return the single tuple $\{x/John\}$ as the result of the derivation. This conclusion is reached due to *John* being the only person with a history of committing murder (see eq. 2.27) and *John* declared as a Person (see eq. 2.19). After the application of UI, the fact *Evil*(*John*) will be asserted to the KB, and any future queries dealing with *Evil* objects will have John included. *Anna* — a parent of *John* — would not be included in this list, as no sentence has declared her having committed murder. UI can be applied many times on universally quantified sentences to derive different consequences.

Existential Instantiation (EI)

$$\frac{\exists v\ a}{SUBST(\{v/k\}, a)} \tag{2.28}$$

"Existential Instantiation is a special case of a more general process called skolemization*"* [29, p.273]: giving a new name to the object which satisfies some condition. This new name is called a *skolem constant* [29, p.273] and must not belong to another object. The sentence a can be replaced by $SUBST(\{v/k\}, a)$ — a new sentence where every variable v is substituted with k — for any sentence a, variable v and constant symbol k; with the condition that k is unique in the KB. We can now *ask* the KB whether there are any evil persons using the query

$$ASK(KB, \exists x Evil(x)) \tag{2.29}$$

and the substitution list of $\{x/John\}$ would be returned. The constant symbol *John* in this case represents a unique name. There is no other person called "John" that is represented using the constant symbol *John*. Once EI is applied to an existentially quantified sentence, the newly substituted sentence can be added, and the original sentence deleted from the KB [29, p.274]. This removal of the old sentence can happen in EI, and not in UI, because in the former case we're only looking for the existence of a single variable that satisfies certain criteria.

2.4.7 Reduction to Propositional Inference (Propositionalization)

Given the tools to perform propositional inference — e.g. model checking — combined with the rules (UI and EI) to infer non-quantified sentences from quantified ones we can transform the problem of FOI to one of PI [29, p.274]. This transformation process works in such a way *"that entailment is preserved"* and is called propositionalization [29, p.274]. All the available propositional algorithms for inference can be applied once all the quantified sentences are transformed:

- *"A universally quantified sentence can be replaced by the set of all possible instantiations"* [29, p.274] after application of the UI rule.

- *"An existentially quantified sentence can be replaced by one instantiation"* [29, p.274] after application of the EI rule.

One problem that exists in FOL that does not occur in PL is the existence of function symbols, as they have infinitely possible ground term substitutions. Consider the function symbol *Mother* with arity 1 taking a single object as argument. Inferring the sentence $Mother(x)$ might never stop as the resulting sentences could resolve to $Mother(Mother(John))$ or with any number of depths using the *Mother* function symbol (e.g. $Mother(Mother(Mother(...)))$). PI algorithms have trouble *"with infinitely large sets of sentences"* [29, p.274]. FOL is for this reason *semi-decidable* [29, p.275] in regards to the question of entailment. This problem is related to the halting problem in the context of Turing machines [29, p.275]. We do not know whether we're stuck in an infinite loop or whether the proof is about to be derived. Additionally, there are algorithms to answer affirmatively to every entailed sentence, but non exist to disprove every non-entailed sentences [29, p.275]. Herbrand's [29, p.274] solution to infinite loops was to incrementally generate instantiations until a propositional proof of the entailed sentence could be constructed.

2.4.8 Generalized Modus Ponens (GMP)

Modus Ponens can be *lifted* from the propositional context to FOL: Generalized Modus Ponens (GMP) [29, p.276]. [29, p.276] defines GMP as the inference rule

$$\frac{p_1\prime, p_2\prime, ..., p_n\prime, (p_1 \wedge p_2 \wedge ... \wedge p_n \Rightarrow q)}{SUBST(\theta, q)} \tag{2.30}$$

Given some atomic sentences $p_i\prime$, p_i, q, and a substitution θ that satisfies $SUBST(\theta, p_i\prime) = SUBST(\theta, p_i)$, for all i, we can derive a new sentence $SUBST(\theta, q)$. This rule formalizes what humans find intuitive. Given the implication

$$\forall x\, Person(x) \wedge Rich(x) \Rightarrow Happy(x) \tag{2.31}$$

and the universally quantified sentence

$$\forall y\, Rich(y) \tag{2.32}$$

we *know intuitively* that $Happy(Anna)$ holds, because *everyone* is *Rich* and $Person(Anna)$ has been established before (see equation 2.20). What we need to establish is that the variables x and y can be substituted with *Anna*: $\theta = \{x/Anna, y/Anna\}$ to make this conclusion obvious for computers as well.

The implication in equation 2.31 is a direct example of the $p_1 \wedge p_2 \wedge ... \wedge p_n \Rightarrow q$ part of the GMP rule (2.30). As such we have

$$p_1 = Person(x) \tag{2.33}$$

$$p_2 = Rich(x) \tag{2.34}$$

$$q = Happy(x) \tag{2.35}$$

From equation 2.19 we know that *John* is a person:

$$p_1\prime = Person(John) \tag{2.36}$$

The implication in equation 2.32 tells us that:

$$p_2\prime = Rich(y) \tag{2.37}$$

Combined we have

$$\theta = \{x/Anna, y/Anna\} \tag{2.38}$$

and conclude that

$$SUBST(\theta, q) = Happy(Anna) \tag{2.39}$$

2.4.9 Unification with *UNIFY*

The process of *"finding substitutions in lifted inference rules that make different logical expressions look identical"* is called *unification* [29, p.276] — a key component for FOI algorithms. *UNIFY* takes two sentences as input and returns a unifier for them if one exists. $UNIFY(p, q) = \theta$ has to satisfy

$$SUBST(\theta, p) = SUBST(\theta, q) \tag{2.40}$$

Unify works by recursively comparing the structures of both sentences, and finding values that *slot in* to the correct position to make both sentences p and q look equal [29, p.277]. In the example

$$UNIFY(Likes(Anna, x), Likes(Anna, John)) = \{x/John\} \tag{2.41}$$

this algorithm finds the non-conflicting substitution $\theta = \{x/John\}$. With the following *UNIFY* call, however, the variables are conflicting

$$UNIFY(Likes(John, x), Likes(x, Danielle)) = \{\} \tag{2.42}$$

since x cannot be *Danielle* and *John* at the same time. By *standardizing apart* [29, p.277] — a fancy way to say renaming all variables to be unique for avoiding name clashes — we can solve the problem:

$$UNIFY(Likes(John, x), Likes(x_1, Danielle)) = \{x_1/John, x/Elizabeth\} \tag{2.43}$$

Now we have non-conflicting substitutions: x_1 is mapped to *John*, and x to *Elizabeth*.

2.4.10 Store and Fetch

With unification in place we can support two essential routines that can be used on a KB: *Store* and *Fetch* [29, p.278]. The former supports assertions of sentences by storing them into the KB: $Store(s)$ where s is a sentence. The latter returns — via $Fetch(q)$ — all unifiers such that the query q unifies with some sentence in the KB. An example illustrates the use of *Fetch*: $Knows(John, x)$ can be interpreted as a fetch request for anyone that is known by *John*. The result of such a request might be $\{x/Anna\}$.

The simplest implementation of *Fetch* can be seen by the following [29, p.278]:

1. Keep all facts in the KB as one long list

2. Given a query q, call $UNIFY(q, s)$ for every sentence s in the list

Even though the performance of such an implementation would be abysmal, it shows simply how a *Fetch* routine might work. To improve on this basic implementation we can add indexing of facts. Thus we can avoid unifications on sentences that have no chance to succeed. In the case where simple indexing of predicates is insufficient due to a predicate-only indexing scheme having long lists to traverse, we can additionally index the arguments along side the predicates as a combined key. [29, p.279]

2.4.11 Forward Chaining in FOL

Forward Chaining (FC), as the name implies, applies *modus ponens* from atomic sentences in a KB, adding new sentences on the way until no more conclusions can be inferred [29, p.280]. FC works by starting from known facts, triggering all the rules whose premises are satisfied, asserting the conclusions. Herein only new facts are considered for assertion, as simply renaming of the same fact are considered duplicates. These are sentences where the semantics are identical, except for their variable names. [29, p.281]

There are three disadvantages to this simple algorithm:

1. Within the core loop, *UNIFY* is used to find all possible unifiers where the premise parts of a rule match with a suitable set of facts in the KB. This pattern matching step is expensive (NP-hard) [29, p.284].

2. The algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even though in most situations only a subset of the whole KB is changed that warrants a starting point for rechecks.

3. Querying the KB with a forward chaining approach results in the generation of facts that are unrelated to the goal.

The system needs to reach a fixed point [29, p.282] of the inference process in order to be able to answer queries. A fixed point is a state a KB reaches, when every inferable sentence has been added to it. [29, p.282]

Given the following atomic sentences

$$Player(Mark) \tag{2.44}$$

$$HitAnkle(Mark, Jason) \tag{2.45}$$

and the implications

$$Player(x) \land IllegalTackle(x, y) \Rightarrow Disqualified(x) \tag{2.46}$$

$$HitAnkle(x, y) \Rightarrow IllegalTackle(x, y) \tag{2.47}$$

$$HitAnkle(x, y) \Rightarrow Injured(y) \tag{2.48}$$

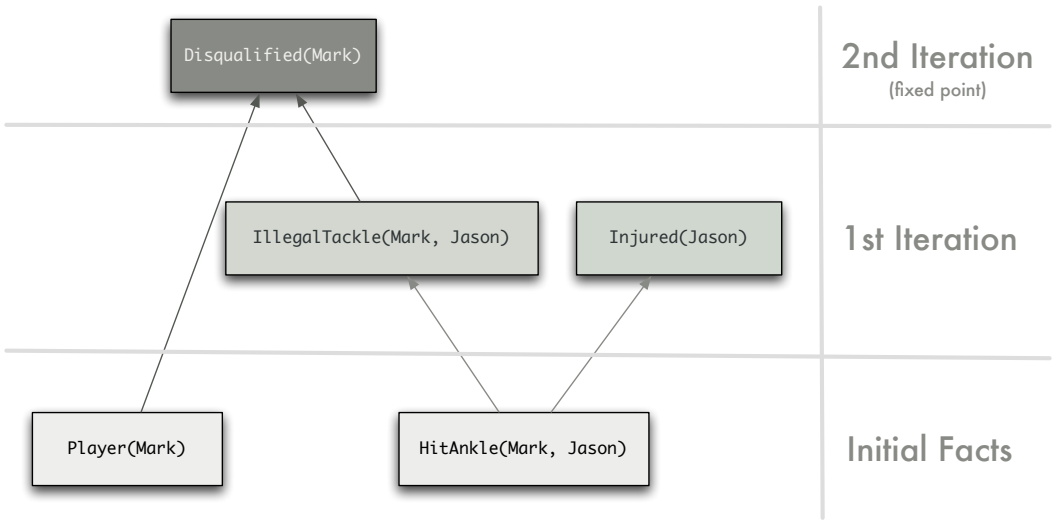

Figure 2.1: Two-Step Iteration Process to infer *Disqualified(Mark)*

we need two iterations of the FC algorithm to reach the conclusion that $Disqualified(Mark)$ holds. In the first iteration we add the following sentences:

$$IllegalTackle(Mark, Jason) \tag{2.49}$$

due to rule 2.47 being satisfied by fact 2.45 with the substitution $\{x/Mark, y/Jason\}$.

$$Injured(Jason) \tag{2.50}$$

due to rule 2.48 being satisfied by fact 2.45 with the substitution $\{x/Mark, y/Jason\}$.
In the second iteration

$$Disqualified(Mark) \tag{2.51}$$

is concluded by satisfying rule 2.46 with the facts 2.44, 2.49 and the substitution $\{x/Mark, y/Jason\}$. Notice that we have added a sentence (2.50), that is irrelevant for the query. This shows the third disadvantage in action. After the second iteration we have reached the fixed point of the KB. Figure 2.1 shows the two step iteration process visually.

2.5 RETE

The RETE algorithm tries to solve some of the three problems outlined in the forward chaining section (2.4.11). The first and second problems are mitigated by heavy caching explained in this section. The second problem is solved by minimizing the number of rules that needs to be re-evaluated due to additions made to the KB. The original RETE algorithm still suffers from the third problem: unnecessary conclusions are still being made during the process of finding the answer to a query, for this the RETE algorithm needs a hybrid approach of doing both forward as well as backward chaining or using a *magic set* [29, p.287] to keep out unwanted conclusions.

The RETE algorithm was first devised by Forgy in 1982 and released under the public domain [41]. Since then many incremental improvements have been researched [42, 43, 44, 45].

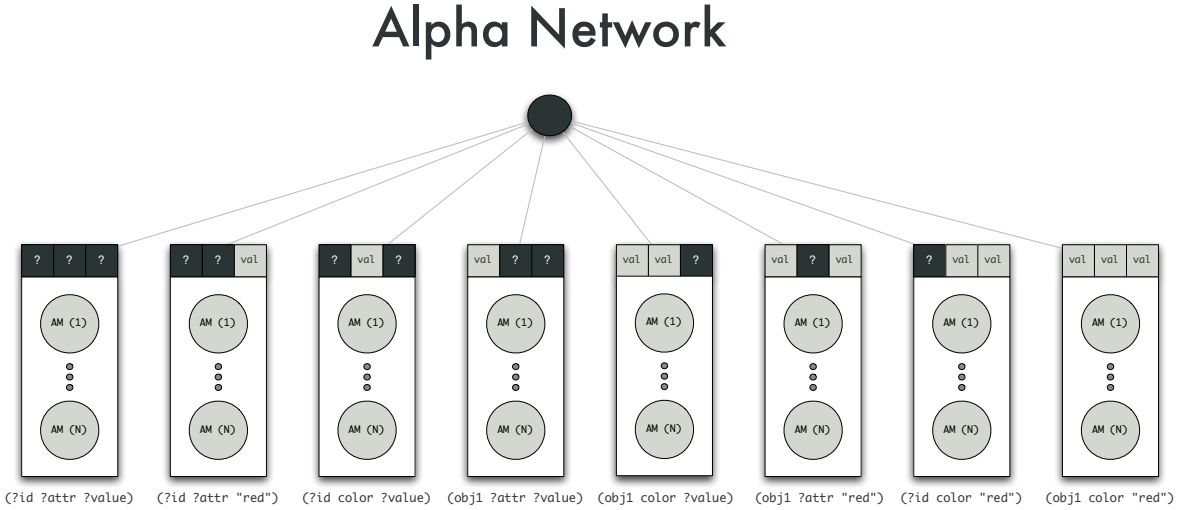

Figure 2.2: RETE Alpha Network with the eight tuple variations

The improvements range from core algorithmic ones to applications in specialized contexts, such as parallel computers. Conceptually, pattern matching algorithms are composed of a fact store — or Working Memory (WM) — on the one hand and a rule store on the other. The former represents all of the knowledge the expert system currently holds. The job of the algorithm is then to identify the subset of facts which trigger the rules stored in the latter.

2.5.1 Working Memory Elements (WME) and Rule Conditions

The core idea of RETE is to build a network of caching nodes for different elements of the pattern matching process. The top most *alpha network*, is a discriminatory data-flow network [46, p.15] designed to exclude as many evaluation paths as possible. The size of this network is determined by the structure of a single fact element — or *working memory element* (WME). One approach of a fact representation is to store a tuple of three elements [46, p.16]. These elements represent the following structure: (*id attribute value*). Conditions — premises or the left-hand side (LHS) of a rule definition — are expressed in the same structure. Variables are prefixed with a '?'. Thus a condition (?*x shape rectangle*) tells us that the *attribute* should match the value *shape*, and the *value*-part should have the value *rectangle*. Anything goes in the *id*-part of the condition in order to trigger a rule that has the preceding condition defined on its LHS. A concrete WME with (obj_1 *shape rectangle*) represents an object that has the *id* of obj_1 and the correct *attribute* and *value* values to activate the fictional rule. Objects can be modeled using this tuple structure by fixing the *id*-part with a consistent symbol. Attributes of the object are modeled as new WMEs with fixed $id = obj_1$ but different attributes and values. A single object is thus depicted as a set of tuples with fixed *id*-part.

From the three element tuple structure, it follows that there are eight possible slots for alpha network nodes. Figure 2.2 shows a sample alpha network with all eight tuple variations.

2.5.2 Alpha Memory Nodes

Connected to the end of the alpha network nodes are the alpha memory nodes. A single alpha memory node represents one condition. The conditions making up a rule then consist

of several alpha memory nodes. If the same condition is shared among several rules, then that single alpha memory node representing that condition is shared among the rules. The goal of this caching layer is to share as many conditions as possible. These nodes serve as connection points between the *alpha network* and *beta network*. The *beta network*'s job is to memoize already evaluated facts matching the respective conditions of the rules. This job is divided among the many different node elements inside the network, achieving a division of labor and separation of concerns.

2.5.3 Beta Network: Join Nodes, Beta Memory Nodes and Production Nodes

Figure 2.3 shows a complete RETE network with its *alpha* and *beta network* components. The first node element is the join node directly referenced from the alpha memory nodes. Its job is to ensure variable consistencies among the facts, which in the end should satisfy all conditions contained in a rule for activation. These consistency checks are driven by the embedded *join tests*. In the original RETE implementation, *join tests* check whether all variable occurrences are consistent so far. Given a rule definition that has conditions ($?car_{id}$ *color blue*) and ($?car_{id}$ *engine v8*) to perform an arbitrary action, then the WMEs (car_1 *color blue*) and (car_1 *engine v8*) satisfy these conditions. The *join tests* for these conditions concern the *id* part of the WMEs. In this case the id car_1 is consistent in both WMEs, and *join tests* succeed in this manner. In the next chapter (implementation section 3.3.4) we define arbitrary tests that are based on these *join tests* at the join node level.

A join node has two parent nodes: the alpha memory node on the right side, and a beta memory node on the left side. An optional production node and a list of beta memory nodes serve as children.

Facts that successfully pass *join tests* are stored as *tokens* inside beta memory nodes. These *tokens* represent already evaluated facts, which not need to be repeated again in subsequent steps. A completed path of successfully evaluated facts end with production nodes, each representing a rule activation with the associated *tokens*. Production nodes are direct children of join nodes. Following the path of a *token* parent/child relationship upwards shows which WMEs — or facts — exactly matched the conditions that triggered the rule associated with a production node.

With each evaluation cycle all activated production nodes are collected in the conflict set. From this point on the *right-hand side* (RHS) of each activated rule is executed. This RHS of a rule is called an action. Actions are domain and application dependent, but the common logic consists of modifying the WM. Each change to the WM sparks a new evaluation cycle.

Orthogonal to the pattern matching problem is conflict resolution [46, p.54], which concerns the order of action execution. In some domains specific rules have higher priorities in regards to their associated actions. The order of their executions is not defined in the RETE algorithm, but is an important component for solving complex problems with expert systems. One way to solve the ordering problem is to add meta data to the rules, called *salience* a positive number ranging from 0 to N, which represents the priority level of rules.

2.5.4 RETE operations

Regardless of the implementation of a pattern matching algorithm, there are elementary interfaces defined in order to modify both the WM and the rule store: adding/removing fact elements and rules. First we define the two different node activation procedures that happen when any of these modifications take place and consequently, rules are evaluated.

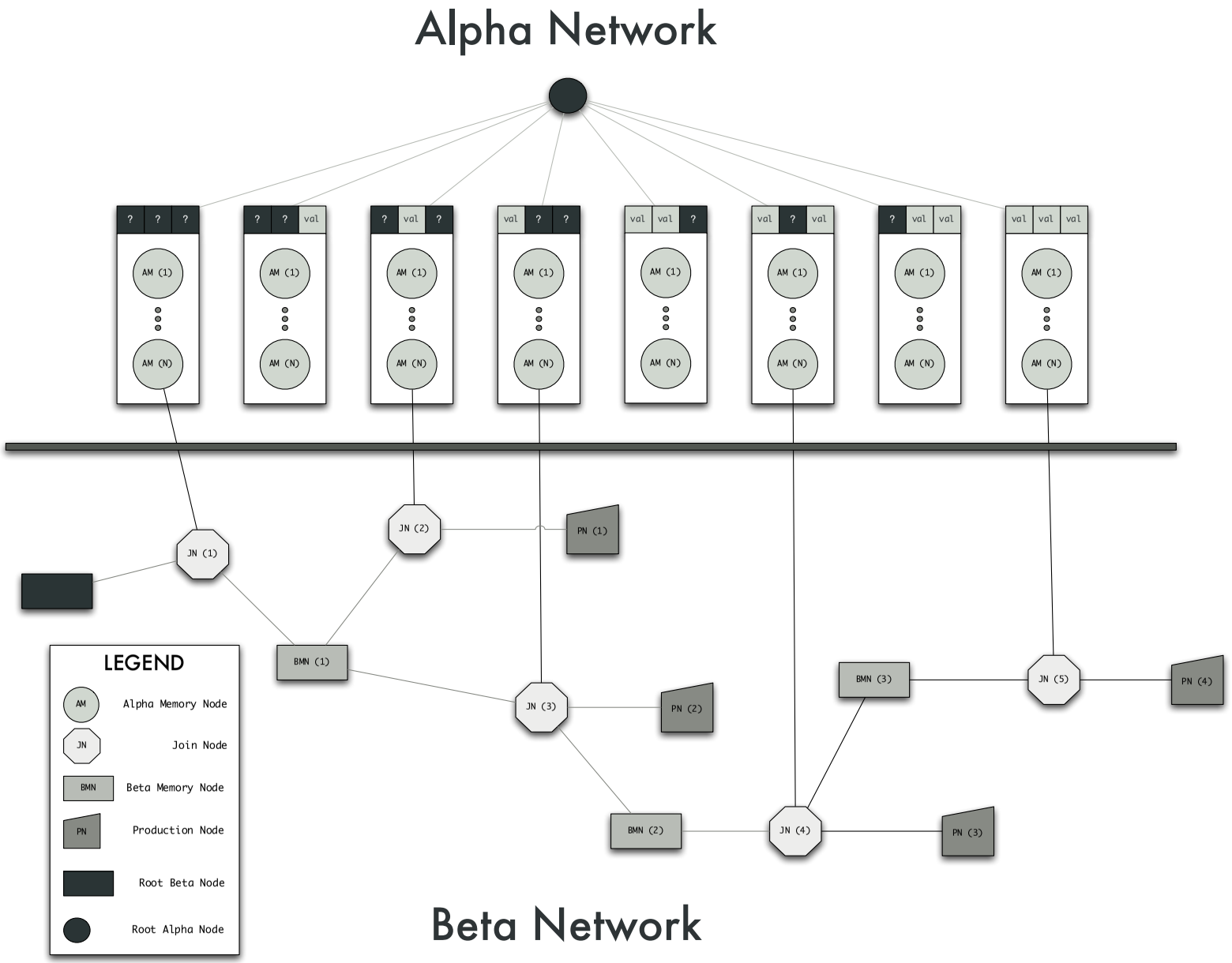

Figure 2.3: Alpha- and Beta-Network

Node Activation (*right-activation* and *left-activation*) What happens during node activation? Depending on the node and triggering parent node we can have these activations:

- join nodes (*right-activation* or *left-activation*): *join tests* are performed, and once successful the optionally associated production node is activated, as well as children beta memory nodes.
- production node (*left-activation* only): production nodes are always triggered by a parent join node. On activation, a new token with the successfully tested WME is added to the production node and the associated rule is triggered. Hereby the RHS part of the rule gets executed.
- beta memory node (*left-activation* only): beta memory nodes are always triggered by a parent join node. On activation, a new token with the successfully tested WME is added to the beta memory node and any children join nodes are *left-activated*.

When do right-activations occur? The only time a right-activation happens is when a join node is triggered for doing so from an alpha memory node. This happens whenever an alpha memory node's associated WMEs are either added or removed. You can see — as shown in Fig. 2.4 — that the recursive *left-activation* nature between join nodes and beta memory nodes lead to the lazy evaluation of rules. Any nodes outside this tree represent rules that hold conditions not required to be re-evaluated.

Adding a new fact element (WME)

The first elementary operation starts with the top of the *alpha network*, in order to find the associated alpha memory node that is responsible for caching WMEs. These WMEs follow a specific pattern defined by the condition stored with the alpha memory node. This is one of the caching effects provided by the algorithm. Existing alpha memory nodes are shared and reused at this level.

In the case of a non-existing alpha memory node, the WME is added to the WM, without continuing the evaluation. Only during rule additions are alpha memory nodes created, and any existing WMEs that have not been evaluated before are checked at that point — via *join tests*.

Once the correct alpha memory node is found, the WME is added to the list of WMEs that the alpha memory node maintains. Children join nodes are then activated with the new WME. This is where *right-activation* — triggered by an alpha memory node — of a join node occurs. *Right-activations* start the cascade of *left-activations* until all children nodes and especially production nodes are activated to trigger rules.

If an alpha memory node already contains a WME where only the *value* part differs, then that WME is overwritten and the usual activation process is triggered. This maintains the consistency of facts within the WM.

Removing a fact element

[46, p.28] outlines several approaches for removing WMEs:

- Rematch-based removal — this approach follows the exact process used for adding WMEs. Instead of following the activation cycle to add *tokens*, we remove them instead. This approach, while elegant, is the slowest, because we do not utilize the information gleaned from the addition process for later removal.

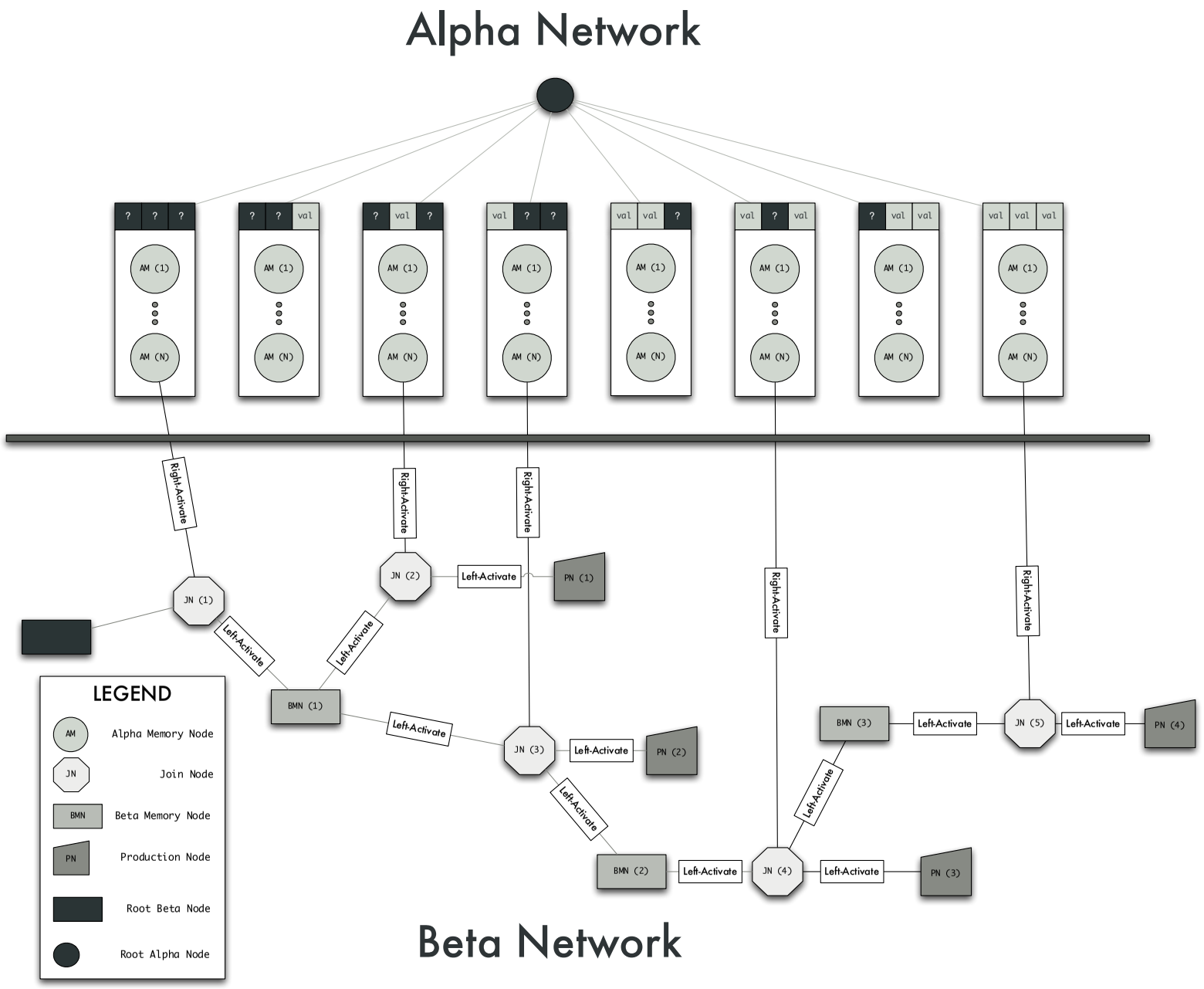

Figure 2.4: Right- and Left-activation of Nodes

- Scan-based removal — this approach exploits the parent/child structure of *tokens* and *join tests* are skipped. *Tokens* inside beta memory nodes, as well as production nodes are tested if any of their *tokens*' parent hold the WME to be removed; if so, a cascading delete is executed.

- List- and Tree-based removal — the general idea herein is to cache the information beforehand, where the WME is stored, and following pointers in order to perform the deletions.

Adding a new rule

The main components required for the RETE algorithm inside a rule are the conditions making up the LHS. For each of these conditions, alpha memory nodes need to be either reused or created, depending whether an alpha memory node responsible for WMEs with the given condition structure already exists. If such an alpha memory node exists, that node will be reused.

From the top — if we start with a new alpha memory node — we start with the root beta memory node. Combined with the *join tests* extracted from the current condition with any earlier ones we can determine to either reuse or create a new join node. Reuse is determined whether a join node already exists with the given alpha memory node, beta memory node and *join tests*. With the join node we either reuse or create a new beta memory node. The next condition in the list uses this beta memory node and starts the same process until all conditions are processed, at which point we add a new production node at the final join node. This production node represents the newly added rule. Triggering this rule happens once an activation cycle touches this production node.

Removing existing rules

For removing rules, we jump to the production node that is associated with the rule. From there we delete upwards any join nodes and beta memory nodes along the path. On deleting a join node, we also check if the associated parent alpha memory node is empty — as in, does not contain other join nodes. If it is indeed empty, we can safely delete it. The other join node parent, a beta memory node can be deleted if it doesn't contain any children join nodes as well. The deletion of a beta memory node is accomplished by removing the associated *token* and deleting the beta memory node in the join node itself. Parent join nodes from the beta memory node are recursively deleted in the same fashion. We stop when we have reached the root beta memory node or we cannot follow the ancestor path anymore due to existing children join nodes.

2.6 Summary

In this chapter we elaborated on the history of *intelligent* programs. Using First-Order Logic (FOL) and Forward Chaining we try to to develop an improved REMAR showing better performance characteristics with the help of the RETE algorithm.

Furthermore we established that the RETE algorithm can solve two of the three problems inherent in the forward chaining algorithm. The RETE algorithm is also a data structure consisting of two networks: alpha and beta networks. They hold nodes with each having specific goals and reasons to exist. Alpha memory nodes serve to cache a single condition. Join nodes perform join tests to make sure consistent variable bindings are observed. Beta

memory nodes maintain the currently partially evaluated conditions making up a rule via tokens. Finally, production nodes serve as a representation of the rule. Activation cycles make sure that each node is evaluated lazily to avoid unneeded rule checks. Rule triggers happen only when production nodes are successfully activated.

Chapter 3

Implementation

3.1 Why are we using RETE?

The previous chapter established the relevance of the RETE algorithm for our purposes, namely in implementing an inference engine along side a DSL that is tailored to our problem domain: synchronization of process instances. [41] introduces the RETE algorithm as the first of its kind that could handle increasing amounts of rules. A literature review highlights the many improvements on the base algorithm that already exists [42, 43, 44, 45], but none in the context of process execution. Furthermore, RETE solves two of the three problems outlined in section 2.4.11, which is important to keep a consistent performance profile.

3.2 How to specify Process Execution Rules?

The model W2 follows consists of two components (1) an inference engine and (2) a DSL. Whereas the former is the focus primarily for programmers, the latter serves as an important user interface language to the inference engine. As such, it requires a simple rule definition language for — possibly non-technical — domain experts. From the programmer's perspective it might be important to know *how* things work inside the engine. In contrast, the domain expert's intention is to use the rule engine to solve problems he/she is facing — concentrating more on the *what*. The domain expert wants the ability to simply manage rules, connecting conditions (LHS) with desired resulting actions to be executed (RHS). This difference in intention is one of the driving forces for the language design, as keeping the language in a high-enough abstraction — removing unnecessary details — is a major concern. One such concern is the problem of optimal condition ordering — important to the programmer, but less so for the domain expert — which we cover in chapter 4. The W2 DSL has a spiritual predecessor: REMAR. Since a DSL already exists defined by REMAR, we try to closely follow its syntax while changing some aspects as needed.

3.2.1 The Anatomy of a W2 Rule

Figure 3.1 shows us the components of a rule definition in the W2 DSL. A single rule definition is commenced with the string *rule*, and the terminator being *endrule*. Everything in-between the *do* and *endrule* build up the action part of the rule (RHS). Following right after the *rule* string is the *rule name*, which we use to semantically differentiate all the rules inside the rule base. The *salience* of the rule determines the priority order in which it is triggered; lower

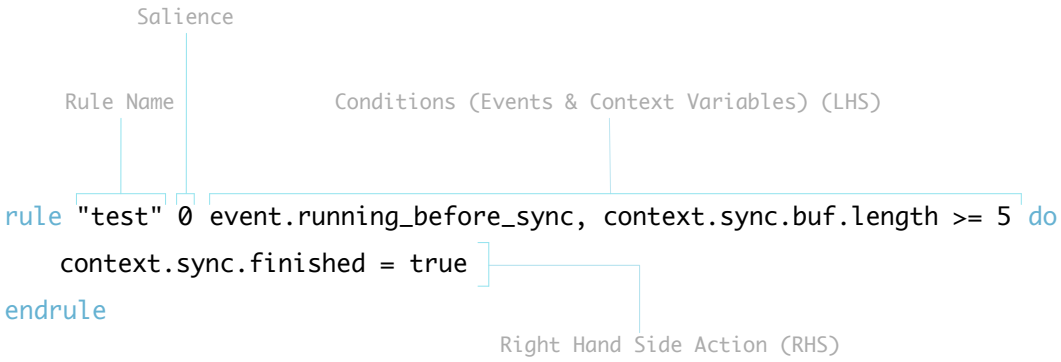

Figure 3.1: Anatomy of a W2 Rule Definition

salience indicating higher priority. The *conditions* part of the rule build up the LHS — made up of *events* and *context variables* — and is used to determine rule activation.

3.2.2 Concepts

Events

W2 does not work in isolation. It has to communicate with other nodes inside the *behavioral shaping network* (Fig. 1.2) to work correctly. W2 runs as an external subscriber to the CPEE listening to its exposed event stream. Through this stream we are listening to two key events: *sync_before* and *sync_after*. Alongside the event type CPEE sends event attributes. Common attributes are

- *instance* to identify the process instance that the event originates from
- *activity* for determining the current activity within the process instance
- *endpoint* to uniquely identify an activity

Events can occur in three different sections within the W2 DSL. (1) At the global document level where we can define which events and associated attributes exist: `event running_syncing_before, instance, activity, endpoint` (2) Within the LHS of a rule definition for testing: `rule "example" 0 event.running_syncing_before do` and (3) within the action part of a rule definition (RHS): `context.sync.buf << [event.instance, event.activity]`. Note that the defined events are used differently depending on use. Within the LHS we are testing for event occurrence in order to trigger a rule, and in contrast the event attributes are looked up in the RHS part of the DSL.

Context Variables

REMAR [9] introduces the notion of *context* — defined as a group or collection of variables $C = \{c_1, ..., c_n\}$ — to solve three issues. First, it allows the representation of internal state. Secondly, for optimization purposes we can exclude some of the rules from the evaluation process by checking only those that are inside the affected context. This saves us from performing unnecessary work. Lastly, a context change can be made transparent to external observers by emitting a *context change event*. This allows subscribers to observe specific context variables and define rules to act on these changes.

Similar to events, context variables can occur in the same three sections of the W2 DSL. At the global document level we can define context variables: `context sync.buf = []`. On the LHS of a rule definition we can perform tests to specify successful rule triggers: `rule "example" 0 context.sync.buf.length >= 3 do`. Finally, on the RHS of a rule definition we can manipulate the variables for assignments or lookups: `context.sync.buf = context.sync.fin`. The specific types a context variable can hold is defined in section 3.3.3. Complex data structures can be modeled using the available types, and as such allows the solving of various synchronization problems.

3.3 Implementation Language

3.3.1 Why are we using Haskell?

The Haskell programming language exhibits characteristics that we deem beneficial for implementing W2. It is classified as a pure, lazy, functional programming language with strong static typing [47]. The *functional* model closely follows the mathematical definition of functions, and with it brings along advantages such as *higher-order functions* (functions that take other functions as parameter), *function composition* and various forms of iteration methods over containers in a homogeneous fashion (*map*, *filter*, *zip*, etc.). Being *lazy* we can model unlimited streams of elements as a function, which simplifies problem solving. On the other hand this model of evaluation departs greatly from established languages, which are *eager*. This can complicate matters in regards to reasoning the performance of applications developed in Haskell. Various tools exist to alleviate this issue, and we exploit the *lazy* characteristic of the language in section 3.3.2. The fixation on staying *pure* is prevalent in Haskell. The direct opposite of this is being *impure* in the sense that we execute side effects inside functions. Being *pure* means being side-effect free and strictly computing the input parameters to calculate its return value. This means that with the one set of input parameters we always return the same output. *Purity* helps Haskell stay relevant in light of the multi-core focus of software development. Being able to utilize additional cores in a CPU will stay important. Real world Haskell programs cannot stay *pure* 100%. Some level of *impurity* needs to be taken into account in order to connect with the outside world (see Fig. 3.2) — as W2 is a subscriber to CPEE, it needs the *impurity* of running network operations to stay in contact with CPEE. *Static typing* seems an unnecessarily complicating matter, as in todays dynamic typing world — via Ruby and Python etc. — *strong dynamic typing* can be achieved. The difference lies in traditional languages using *weak* static typing (C, C++, Java), which is the source of many bugs and costly security flaws (buffer overflows, null pointer exceptions, and *off by 1* errors). With *strong static typing* most of these issues are non-existent and through experience with the Language we can say the compiler can serve (1) as a safety net for developing applications to avoid most errors and (2) as a documentation for defined data types and functions, which are helpful for continual software maintenance.

One of the disadvantages for Haskell — due to the *functional* nature — is the inability of changing references to variables, which are traditionally solved using pointers in languages such as C [48]. This feat cannot be accomplished by the core language, requiring special libraries to be used; We will touch on this issue in section 3.3.2.

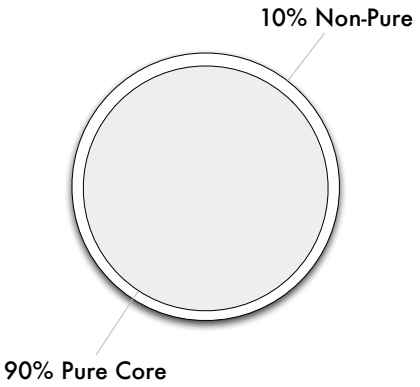

Figure 3.2: Purity inside Haskell Programs

Parallelism and STM in Haskell

Concurrent applications based on locks and control variables exhibit deficiencies. These include:

- relying on conventions without compiler support — C code in the Linux Kernel handle lock problems inside comments that show the conventions that should be followed. No compiler-backed support is available for such an approach.
- deadlocks — due to processes locking on resources and keeping each other out to accomplish their jobs
- non-composable — once functionality have been implemented based on locks, these functions cannot be reused inside other functions. Such a drawback violates many good software principles, including *Don't Repeat Yourself* (DRY), modularity and separation of concerns.

Haskell has several primitives for both parallelism and concurrency [49, 50]. With STM, Haskell solves the listed deficiencies above allowing robust, composable and compiler-supported functions to be deployed for a multi-core context. [51, Ch.24] shows that composable atomic functionalities are central for developing concurrent software.

[52] observe that transforming a sequential data structure into a multi-core context via STM results in significant speedups. [53] show that a IORef *check and set* method shows the best results. A modified STM approach show second best results. IORef and MVar based approaches show minimal scaling in a multi-core context.

Strictly translating the data structure with STM is not sufficient. [54] show the problems with parallel rule activations in RETE. In order to avoid such problems we have devised an *island-based* design of the RETE data structure. In contrast to a traditional RETE graph we introduce *islands* that group those Alpha Memory Nodes together that are shared among several productions. With this, an Alpha Network can have many islands and thus can hold several separate Beta Networks at a time, allowing independent rule firing for each island. The following optimization chapter leaves out the benchmarking of this island-based data structure as the time budget allowed only the optimization of the *conjunct ordering* problem in our process execution domain. The island-based data structure is still used for the implementation excluding the logic of periodically merging and splitting of islands which is required to have disparate and independent islands. A comprehensive comparison for a

multi-core context would require several (IORef-based, MVar-based, etc.) implementations that we left out and could be the subject of future work.

3.3.2 Underlying Data Structure

As touched shortly beforehand, we encountered difficulties in modeling the underlying data structure of the RETE graph. This is due to Haskell's *functional* and *pure* nature and can be seen in the immutability of data structures and the non-existence of pointers. The latter functionality exists in Haskell — via IORef, STRef or MVar — but with the cost of losing the *pure* nature sacrificing the ability to cope with better scaling characteristics in parallel and concurrent execution of the application.

In a functional setting every state change has to be accomplished by copying the whole state, changing the intended focus point and returning that new state. In the simple example of a list, we might have a list of three integer elements as the state: [1,2,3]. Changing this list state to [1,2,4] in Haskell generally means recursively traversing the list from the head while collecting the observed elements until the intended element is reached. At which point instead of returning the element 3, a new element is returned, which is 4. The result is a new list with the old elements intact, except with the new value 4 instead of 3. Even though semantically a new copy of a list has been generated, internally most functional languages — including Haskell — optimize such changes by only modifying the minimal subset of the whole list. In contrast, with pointer-based languages we are able to pinpoint exactly which element we want to change. In C, this is done by referencing the memory address of the variable we intend to change.

Requiring the processing of the whole state in order to perform small changes can naturally lead to performance issues. The following subsections elaborate some of the methods we've explored that allow us to base our RETE data structure in a strict *pure* manner allowing us to exploit the advantages in a multi-core setting.

Zipper

The first approach for the underlying data structure was based on Huet's Zipper [55] data structure. It works by deriving an auxiliary navigational data structure from another, using a focus point pinpointing a *current* element within the global state. The Zipper works similarly to the list modification example. Although with this approach we do not explicitly process the whole state. Changing a single focus point means traversing on the Zipper state using directions — such as left, right, up, and down — and once reaching the intended point, an application-specific swapping operation is performed to switch out the element with a new one. Once deserialized, the Zipper data structure can be transformed into the original data structure with the intended changes already complete.

With our implementation we tried navigating the resulting Zipper state after modeling our data structure to be compatible with the Zipper. The major problem that it exhibited was of not being able to traverse the parent relationships — if there are more than one — from any given node. We concluded that Zippers — at least the Zipper library we've chosen [56] — only support homogeneous data structures. The model we have in order to implement the whole RETE graph consists of several distinct nodes embedded within. As the RETE data structure we modeled is a heterogeneous graph, we could not realize the underlying data structure with the help of Zippers. Figure 2.3 in the last chapter shows the different Beta Network nodes and their references to parent nodes, establishing its graph and non-homogeneous nature.

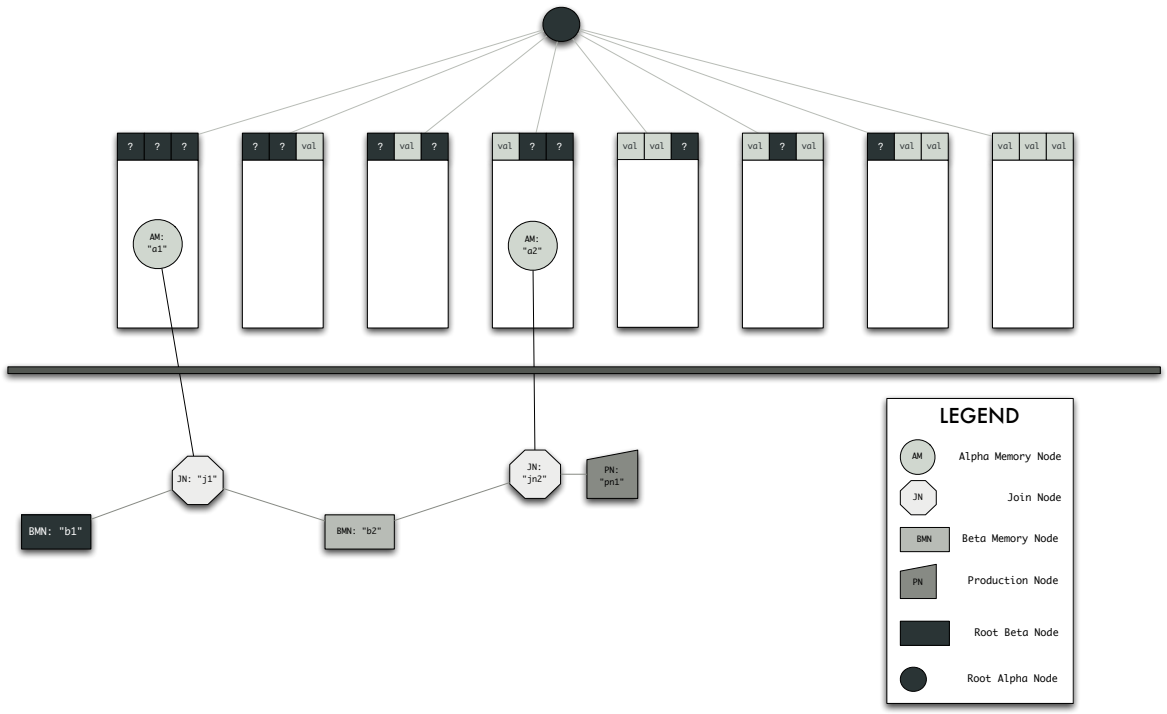

Figure 3.3: The example RETE Network in graph-form

Knot tying

This approach is pure as well. The basic Idea stems from the use of Haskell's laziness feature whereby future values are referenced at the point where they do not yet exist. Only after *the knot has been tied* are the values correctly evaluated and available for access [57]. With this approach we can model the RETE data structure as a list of serialized nodes with reference ids as shown below, which is equivalent to the graphical version depicted in Fig. 3.3:

```
[("island", IFBetaIsland "b1" ["a1","a2"])
,("b1", IFBetaMemory "j1")
,("a1", IFAlphaMemory "j1")
,("a2", IFAlphaMemory "j2")
,("j1", IFJoinNode "b1" "a1" ["b2"] None)
,("b2", IFBetaMemory "j2")
,("j2", IFJoinNode "b2" "a2" [] "p1")
,("p1", IFProductionNode)
]
```

Each element within the list represent a tuple of a reference id and a node description, with the parameters being either id references to the parent or children nodes. Non-referential attributes can be represented inline. Actual node instances are created with the help of the node description. Although after a transformation we have our generated data structure, it is not yet possible to use such a serialized list structure as the basis for performing changes to the RETE data structure. What we have is a *list-form* of the data structure. We call the final RETE data structure to be in the *graph-form*. Between the *list-form* and the *graph-form* we have an intermediate hash table of instantiated nodes lazily referenced via *knot tying*. In this

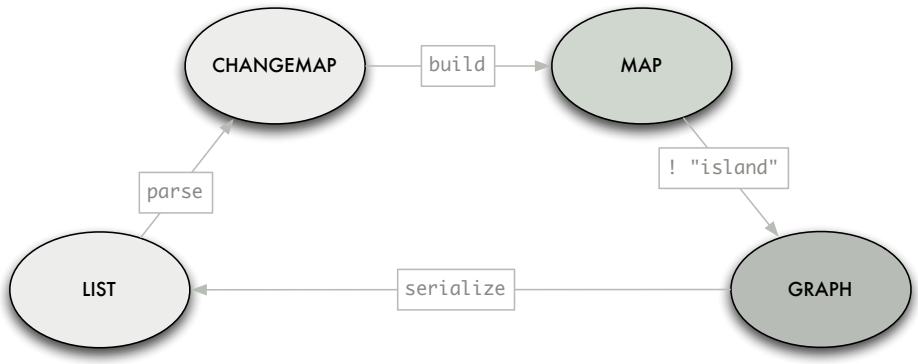

Figure 3.4: Various *forms* of the knot tying data structure approach

map-form we can extract the final data structure (in *graph-form*) by looking up the index of the island. The *map-form* cannot be used as the basis for changing, as at that point the *knot tying* process has been finished, and changes to node instances that have live references are not propagated.

In order to support change, we introduce the *change-map* form, which is directly generated from the *list-form*. This is accomplished by linearly traversing the *list-form* and inserting the unique id as key, and the node description as value. As such, direct changes to a node description can be accomplished by accessing its reference id. Fig. 3.4 shows the model of the different forms of the data structure using the *knot tying* approach. The flow starts with an island instance — in *graph-form* — that is *serialized* to the *list-form*. A *changemap-form* is generated from the *list-form*, and a *map-form* can be *built* by instantiating nodes using the node descriptions. Once the *changemap-form* is derived it can serve as the basis for changing the island graph. This mechanism is used whenever an island graph is changed and thus occurs:

- during right-activation of Alpha Memory Nodes, which happens when adding or removing WMEs.
- during rule additions or removals

Each *form* has its purposes. The *list-form* and *change-form* serve as description of the actual RETE data structure or graph, and the latter being exclusively used for change. The *map-form* is used for holding the node instances generated from the previous two *forms* and keeping the correct references between the nodes using *knot tying*. Finally, the *graph-form* is extracted from the *map-form* by looking up the *island* key. As each form has to be transformed during a single change, memoizing the *change-form* for future changes saves us the trouble of having to repeatedly serialize the *graph-form* and improve performance as well.

3.3.3 Value Types

The supported types — which a value can hold within conditions as well as *WMEs* — are summarized with the following value data type definition: *Val*. It loosely follows the JSON specification of supported value types, and as such allows flexible rule definitions. Beside the primitive data types, we support container types: *ValList* for list types and *ValMap* for dictionary type values. With these container types we allow tests on such data types

(e.g. determining whether a list has certain length, or a dictionary includes a specific key or value) on the condition level, as well as in stored facts via *WMEs*.

```
data Val = ValInt    Int
         | ValString String
         | ValFloat  Float
         | ValDouble Double
         | ValBool   Bool
         | ValList   [Val]
         | ValMap    [(Val, Val)]
```

3.3.4 Handling arbitrary tests

There are at least two ways for implementing arbitrary tests; one way is to extend the Alpha Network, and another is to use the existing Join Nodes and extend their matching logic to include arbitrary tests [46, p.17]. Our implementation is based on the latter approach. We extend the join tests by optionally specifying the type of join test to perform. With *default join test* we utilize the default join test logic.

```
getJoinTestsFromCondition :: Condition -> [Conditions] -> [JoinTest]
```

On adding a new rule, the conditions are subjected to the *getJoinTestsFromCondition* function which takes the current condition and all earlier ones to generate a list of *JoinTest* instances. These work in a backward fashion. With relative indexing the function fetches the nearest condition that is relevant for *join testing*. The *JoinTest* instances are then used to perform join tests on the current WME and relevant tokens associated in the *BetaMemory* Nodes. This join test is performed during right- as well as left-activation of the appropriate nodes. There are three *JoinTest* types: *DefaultJoinTest*, *VariableJoinTest*, and *ConstantJoinTest*.

```
data ConditionField = IdentifierField
                    | AttributeField
                    | ValueField

data DefaultJoinTest = DefaultJoinTest { jtFieldOfArg1 :: ConditionField
                                       , jtConditionOfArg2 :: Int
                                       , jtFieldOfArg2 :: ConditionField
                                       , jtComparator :: Comparator
                                       }
```

The *default join test* is performed when none of the special join tests are specified (which handle the arbitrary tests). In this variation the *getJoinTestsFromCondition* function extracts *DefaultJoinTest* instances by finding variables, which occur more than once. The goal here is to make sure that variables with the same names within a production rule are bound to the same values. The *jtComparator* attribute is always set as the *equal (==)* Comparator. The *jtFieldOfArg1* and *jtFieldOfArg2* specify the fields in which the variable can be found within the current as well as in one of the earlier conditions respectively. *jtConditionOfArg2* tells us the relative index in which *Token* instance the value lookup should be performed.

```
ConstantJoinTest { jtFieldOfArg1 :: ConditionField
                 , jtComparator :: Comparator
                 , jtConstantValue :: Val
                 }
```

Constant join tests are one of the extra join tests that can be optionally specified. As the name suggests the *constant join test* performs a comparison between the current *WME* instance with the value that is specified in *jtConstantValue* on the field that is specified in *jtFieldOfArg1*.

```
Rule "sample rule" 0
     [ createCondition "?joe" "pos" (VVar $ Var "joe_pos")
                       [ ConstJTest (Var "joe_pos") equal (ValInt 2) ]
     ] ( ... arbitrary RHS action ... )
```

With the preceding rule definition a *constant join test* would be performed on the *ValueField* (because the *ValueField* holds the variable *?joe_pos*). The constant value that makes the *join test* pass is the integer value *2*. During *left-* and *right-activation* all *WME* instances matching this condition of the rule are tested for the value field to contain the value *2* as an integer.

```
VariableJoinTest { jtFieldOfArg1 :: ConditionField
                 , jtConditionOfArg2 :: Int
                 , jtFieldOfArg2 :: ConditionField
                 , jtComparator :: Comparator
                 }
```

Variable join tests work in a fashion that is similar to *default join tests* — as seen by the similar definition of the attributes. They both look at preceding tokens in order to perform the join test. The only difference is that the chosen fields are determined by the specfied variable. Consider the rule definition:

```
Rule "sample variable join test rule" 0
     [ createCondition "?x" "age"    (VVal $ ValInt 20) []
     , createCondition "?z" "height" (VVar $ Var "d")
         [ VarJTest (Var "z") equal (Var "x") ]
     ] ( ... arbitrary RHS action ... )
```

With the preceding rule definition a *variable join test* is performed on the *identifier* part with the *identifier* part of the preceding condition. The *identifier* portion has been chosen due to the variable x being in the *identifier* field of the condition. With two fact tuples (*WMEs*) added to the *WM*: *("christian" "age" 20)* and *("christian" "height" 180)* the *variable join test* would succeed, as the *identifier* parts of both are identical. The *VariableJoinTest* instance would be populated with the following values:

```
VariableJoinTest { jtFieldOfArg1     = IdentifierField
                 , jtConditionOfArg2 = 1
                 , jtFieldOfArg2     = IdentifierField
                 , jtComparator      = equal
                 }
```

Comparators are trivially implemented. Beside a description, it requires a function that takes two value inputs and returns a boolean result. As such the *equal* comparator can be implemented as:

```
data Comparator = Comparator { cDescription :: String
                             , cFunction :: Val -> Val -> Bool
                             }

equal :: Comparator
equal = Comparator { cDescription = "(==) comparator"
                   , cFunction = \sym1 sym2 -> compare sym1 sym2 == EQ
                   }
```

The inference engine natively supports the following comparators: *lessThan*, *greaterThan*, *lessEqualThan*, *greaterEqualThan*, *equal*, *notEqual*, *modifiedEqual* [1], *listLength*, and *listEmpty*.

3.4 DSL Compilation Process

Having implemented a RETE-based inference engine in Haskell we shift our focus to the W2 DSL. In section 3.2 we explained the goals, design and concepts of the DSL. In this section we will dive into the compilation process of the DSL to make it run on top of the RETE engine.

Applications that deal with language follow the general process outlined by [58]. First, the input is processed through a *Reader* component that transforms the input into an *internal representation* (IR). This *IR* may be an *abstract syntax tree* (AST) or any other host-native data structure. Subsequent passes — the actual number being application and domain specific — work on the input and update the *IR*. At the end, the *Generator* component transforms the *IR* into the desired output. For most compilers this is native machine code. In our case, we hope to transform the W2 DSL into native Haskell code, that is being executed on top of the inference engine.

Traditionally [59] DSL design decisions are made in collaboration with domain experts in order to synchronize the understanding of the problem domain. Our influence was the existing DSL implemented in REMAR [9]. Our goal was to develop an identical DSL that could be ported from REMAR to W2. Some differences exist mainly in the RHS portion of the DSL, which we elaborate in the following subsections.

Rule definitions in the native implementation (Haskell) as shown below exhibit a major disadvantage: *it is too verbose*:

```
Rule "rule test" 0
     [ condition "curr_evt" "name"     (VVal $ ValString "before_sync") []
     , condition "curr_evt" "instance" (VVar $ Var "event_instance") []
     , condition "curr_evt" "activity" (VVar $ Var "event_activity") []

     , condition "?ctx_buf_sym" "type" (VVal $ ValString "context") []
     , condition "?ctx_buf_sym" "name" (VVal $ ValString "buf")     []
```

[1] *modifiedEqual* is a comparator that is a higher-order function that accepts an arbitrary modifying function on the second value. As such the second value can be modified before the comparison takes place.

```
, condition "?ctx_buf_sym" "value" (VVar $ Var "ctx_buf_value") [
   [ ConstJTest (Var "ctx_buf_value")
                (tListLength ">=")
                (ValInt 5)
   ]
]
# some RHS action here
```

Verbosity in this context has a negative impact to the productivity of the rule writer, as each read through is impacted by the longer syntax. Full comprehension of the rules is taxing to the mind, as it tries to pick out only the relevant parts of the syntax for the current domain problem at hand. [60] talks about such productivity issues of DSLs: *"A DSL offers appropriate domain-specific notations from the start. Their importance should not be underestimated as they are directly related to the productivity improvement associated with the use of DSLs"*. Domain-specific notations allow us to keep out the irrelevant parts of the Haskell Rule definition, which is important for a domain expert working with W2. These include:

- The order of conditions. The effects of condition ordering is explained thoroughly in the next chapter (4).
- The internal IDs which represent the events and context variables: "curr_evt", "?ctx_buf_sym"
- The typing information for each condition value (e.g. VVal $ ValString "before_sync").
- The types of join tests (e.g. *ConstJTest*, *VarJTest*)
- The position of extra join tests, which have to occur at the last condition where the lookup can be performed. This is something that the compiler should do for us automatically.

The components that **are** relevant and required to be distinguished from the irrelevant parts are the following:

- The rule name (e.g. "rule test")
- The salience value for rule activation order. (e.g. 0)
- The distinction between events and context variables
- The actual event or context variable names (e.g. "before_sync", "context.buf")
- The attribute lookup (e.g. *tListLength*, used to lookup the length of a list)
- The comparison operator (e.g. ">=")
- The target value (e.g 5)

The same rule semantics of the last example rule can be defined more succinctly with the following W2 DSL. Notice that the core elements still remain and the unnecessary details are eliminated from distracting the reader.

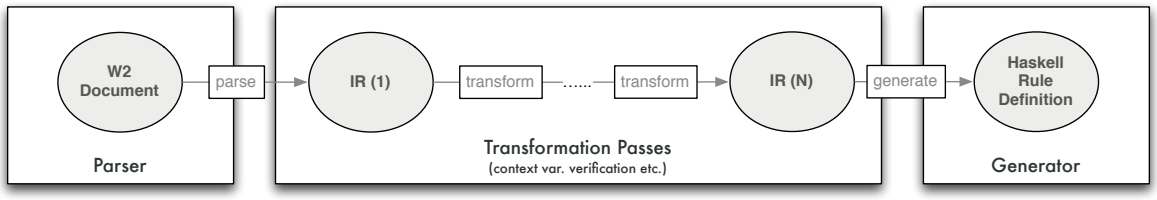

Figure 3.5: W2 DSL Compilation Process

```
rule "rule test" 0 event.before_sync, context.buf.length >= 5 do
    # some RHS action here
endrule
```

What is now required is a compiler that translates the preceding W2 DSL into the Haskell native definitions. [58] defines such a compiler among the different types as a *source to source* compiler. In contrast to native compilers ours does not emit native machine code, and unlike *cross compilers* we do not try to emit code that runs on different hardware platforms. Since our target language is Haskell — another high-level language — the compiler can be classified as a *source to source* compiler. As one of the intermediate passes, our DSL compiler will apply optimizations. These optimizations are the subject of chapter 4.

The workflow of our DSL compiler follows this process (depicted in Fig. 3.5):

1. The whole W2 DSL source document is parsed using a monadic parser combinator library in Haskell called *Parsec*.

2. Individual document elements are parsed: context variable definitions, event names, and rules.

3. The rule parsing logic is divided into LHS and RHS. For the LHS, only the conditions, rule name and salience are extracted. For the RHS another custom compiler is invoked to translate the inner DSL. This is the part where we deviate from REMAR. Whereas REMAR follows a Ruby-based embedded compiler approach, which allows arbitrary Ruby code to be executed as the RHS, we implement a custom stack-based RHSDSL that lets the user to:

   - manipulate context variables
   - perform specific and constrained actions, such as calling external web service calls

4. The *IR* is translated to a native Haskell-based rule definition.

3.4.1 LHS and W2 Document Parsing

The following grammar and parsing function definitions in Haskell handle the LHS — conditions, rule name and salience — as well as the global W2 Document sections — including event and context variable definitions. Each section of the document are turned into the *IR*, which in our case are native Haskell data structures (e.g. *W2Document*, *W2ContextDef* etc.).

```
document :: Parser W2Document
document = sepBy1 documentSection (many1 newline)
```

```
documentSection :: Parser W2DocumentSection
documentSection =  comment
               <|> eventDefinitionWithArgs
               <|> eventDefinition
               <|> contextDefinition
               <|> ruleDefinition

comment :: Parser String
comment = skipMany (char ' ') >> char '#' >> line

eventDefinition :: Parser W2EventDef
eventDefinition = EventDef
               <$> (string 'event')
               <*> (many1 space >> atom)


contextDefinition :: Parser W2ContextDef
contextDefinition = ContextDef
                 <$> (string 'context' >> many1 space *> assignments)

assignments :: Parser [Assignment]
assignments = sepBy1 (wrappedInSpaces assignment) (char ',')

ruleDefinition :: Parser W2RuleDef
ruleDefinition = Rule
    <$> (string "rule" >> spaces >> parseString)
    <*> (spaces >> parseSalience)
    <*> (spaces >> parseConditions)
    <*> (Action
        <$> parseRHSAction)
...
```

Each monadic function definition focuses on one small part of the grammar, allowing flexible reuse throughout the parser definition.

3.4.2 RHS Action Interpretor

The parsing of the RHS part of rule definition is handled via *parseRHSAction*. As the W2 DSL is not an embedded [59] but an external one, we do not have the luxury of evaluating the RHS part as native Haskell code to execute arbitrary application logic. The goal of the RHS is to enable the user to (1) manipulate context variables and (2) call external web services — including pushing the *continue* or *stop* vote to the process engine for manipulating running process instances. Allowing the user to execute arbitrary code falls outside these use-cases and the restricted environment serves as a safe sandbox.

We achieve this sandbox by implementing a *stack-based interpreter*. The RHS action part of the DSL is parsed and transformed to byte code that is then executed on top of it. [58] outlines two differing interpreter styles with which languages are executed. The *stack-based* approach follows a *last in first out* (LIFO) stack model with its standard operations: *push*

and *pop*. Commands — byte code or operations which we use interchangeably — are *pushed* to the stack. Execution happens by taking from the stack — via *pop* — the current command and running it. Most commands reference other elements within the stack which are *pop*ed as required. Results are then stored by *push*ing it to the stack again. Through this — potentially endless — cycle of *push*, *pop* and *execute* of commands our RHS actions can be run inside the sandbox. A *register-based interpreter* explicitly models a number of registers, referenced in each command. The advantage of the *register-based* model is the lower number of operations that have to be executed compared to the *stack-based* one. Most languages used in industry — such as Java and C# — are based on the *stack-based* approach for their virtual machines. Some languages, such as Lua [61] use both models. For the execution, the language utilizes the *register-based* approach, but the programmer API is exposed as a *stack-based* model, which is easier to work with. We chose the *stack-based* approach due to ease of implementation and natural modeling of the supported commands in Haskell. Having a byte code *stack-based* interpreter allows us random action generation, which is relevant for the next chapter (4): optimization.

The following example illustrates the process of parsing the RHS action part of the DSL, which ends as executable byte code for the *stack-based interpreter*. The W2 DSL snippet

```
context.buf << [event.instance, event.activity]
```

semantically means *adding the list `[event.instance, event.activity]` to the context variable `buf`*. It uses both *list construction* and *list append*. After parsing this snippet, its *IR* is represented as

```
W2AListAppend (W2AContextName "buf")
              (W2AList [W2AEventLookup "instance"
                       ,W2AEventLookup "activity"])
```

At the *transformation* stage of the compiler, that *IR* is then translated into the following interpreter operations

```
[ Push $ ValString "activity"
, Call EventLookup
, Push $ ValString "instance"
, Call EventLookup
, Call $ ListConstruct 2
, Push $ ValString "buf"
, Call ContextLookup
, Call ListAppend
]
```

This resulting byte code is directly executable on top of the *stack-based* interpreter within the sandbox. The W2 DSL syntax supports the following operations for the RHS actions:

1. Dictionary lookup, list construction and list append

```
context.sync[event.endpoint] << [event.instance, event.position]
```

2. Assignment (on context variables)

```
context.sync.step = :finish
```

3. *WME* lookup, list pop, and custom functions (such as *continue*)

```
continue context.sync.buf[:c].shift
```

4. List traversal with function application (e.g. apply a function to each element of a list via *each*)

```
context.sync.fin.each continue
```

5. List deletion

```
context.sync.run.delete [event.instance, event.position]
```

6. If statements

```
if context.finished then
    ...
else
    ...
end
```

3.5 Summary

In this chapter we introduced the implementation language Haskell, which we use for writing the RETE-based inference engine and the DSL compiler. The language exhibits unique characteristics, namely *purity* and its *functional* nature which lead us to the way of researching and prototyping different models to base the RETE graph on. We explained notable implementing issues during the development of RETE.

We also introduced the W2 DSL, that simplifies rule specification and definition for domain experts, who are not concerned with the minutiae of the Haskell and RETE-backed inference engine. The W2 DSL follows the generic language transformation process of (1) parsing in the document (2) storing the whole program in an *IR* and finally (3) translating the *IR* into the Haskell-based rule definitions, which can be executed directly with the W2 inference Engine. Step (3) is further divided into the LHS and the RHS part. The latter (Action part of the rule) being implemented with a byte code *stack-based interpreter* to enable a safe sandbox of action execution and to enable optimizations, which is a topic of the next chapter.

With a RETE-based inference engine and a DSL in place, we are able to implement the synchronization algorithms implemented in REMAR. Other domains can benefit from this model by adapting the grammar as well as actions to a more domain-friendly syntax to allow productive usage of the language.

Chapter 4

Optimization & Benchmarking

We noticed that the same rule base can be defined differently in terms of the order of conditions within the rules, which has an impact to the matching performance. As this is a design decision that has to be made for the DSL Compiler and existing literature hinting to various heuristics (see next section 4.1) that have been extracted from their observations on this issue, we chose to empirically determine the optimal condition ordering for our domain: process execution. In this chapter we will dive into the specific optimizations that are made possible, introduce the benchmarking methodology we pursue and compare the performance between (1) unoptimized RETE vs optimized RETE and (2) unoptimized RETE vs brute force naive forward chaining rule matching.

4.1 Process Execution specific RETE optimizations

The rule conditions are composed of events and context variables, as explained in 3.2.2. They are both further divided into subcomponents. Whereas the event name and event attributes make up the event conditions, the context variable conditions consist of the context type, context variable name and context variable value. The creation of these conditions are controlled by the existence of their respective events or context variables on either the Left Hand Side (LHS) or the Right Hand Side (RHS) of the rule definitions. The LHS of the following rule definition — in W2 DSL syntax — holds references to the event `running_before_sync` and the context variable `buf`, whereas the RHS holds a reference to the context variable `finished`.

```
rule "test rule" 0 event.running_before_sync, context.buf.length >= 5 do
    context.finished = true
end
```

Combined, the rule definition above is compiled to the following conditions. Notice that context variables occurring on the RHS never have any join tests associated with them. They only have to exist in the condition list, for the variable lookup mechanism to work. In this case the variables `event_instance`, `event_activity`, `ctx_finished_value` and `ctx_buf_value` are available for lookup and manipulation.

```
[ -- event "running_sync_before"
  condition "curr_evt" "name" (VVal $ ValString "running_before_sync") []
, condition "curr_evt" "instance" (VVar $ Var "event_instance") []
```

```
  , condition "curr_evt" "activity" (VVar $ Var "event_activity") []

    -- context variable "finished"
  , condition "?ctx_finished_sym" "type" (VVal $ ValString "context")  []
  , condition "?ctx_finished_sym" "name" (VVal $ ValString "finished") []
  , condition "?ctx_finished_sym" "value" (VVar $ Var "ctx_finished_value") []

    -- context variable "buf"
  , condition "?ctx_buf_sym" "type" (VVal $ ValString "context")  []
  , condition "?ctx_buf_sym" "name" (VVal $ ValString "finished") []
  , condition "?ctx_buf_sym" "value" (VVar $ Var "ctx_buf_value")
      [ ConstJTest (Var "ctx_finished_value")
                   (tListLength ">=")
                   (ValInt 5)
      ]
  ]
```

[30] state the various effects of condition ordering on the RETE matching time, confirmed by [62]. The "*place restrictive conditions first*" heuristic refers to the first effect of condition ordering, wherein the number of created tokens for partial production node instantiation depends on that ordering. Following the heuristic reduces the token count inside the beta network and thus shortens the time required to evaluate said graph. In the process execution domain the event conditions seem at first glance to satisfy the heuristic, as they tightly group all those rules together with the same event. The question remains open, as to which specific part of the event conditions are considered more *restrictive*: the name condition or the event argument conditions?

Another mentioned effect of condition ordering in [30] is the link between dynamic WMEs and the matching time. In contrast to dynamic WMEs, static WMEs are those fact elements not modified during the lifetime of the alpha and beta network. Thus dynamic WMEs are fact elements constantly removed and added — possibly with differing values. The time to remove WMEs depends on the chosen WME removal algorithm (see 2.5.4) and can be appropriately optimized. The cost of WME addition on the other hand is always incurred and cannot be avoided for dynamic WMEs. The suggested position for conditions holding dynamic WMEs is at the end of the condition list, minimizing the effect of WME changes to subgraphs by keeping them as small as possible. The categorization of dynamic WMEs in this domain is not clear-cut, as all events and context variables are possibly dynamic. The current event is always changing, and context variables can always be modified on the RHS.

Nayak et al [63] state that domain specific heuristics need to be employed for the question of optimal condition ordering. In any case, domain experts tasked to build rule bases should not be faced with this question. It is a task that should be hidden from the DSL user and handled automatically by the rule compiler and interpreter.

We thus try to empirically find the optimal condition order in the context of process execution.

4.1.1 Rule Types

Before we move on to the benchmarking methodology employed for this endeavor, we will classify the different types of rules that can occur in the context of process execution. Beside the existence of events or context variables, we further observe whether any of the context

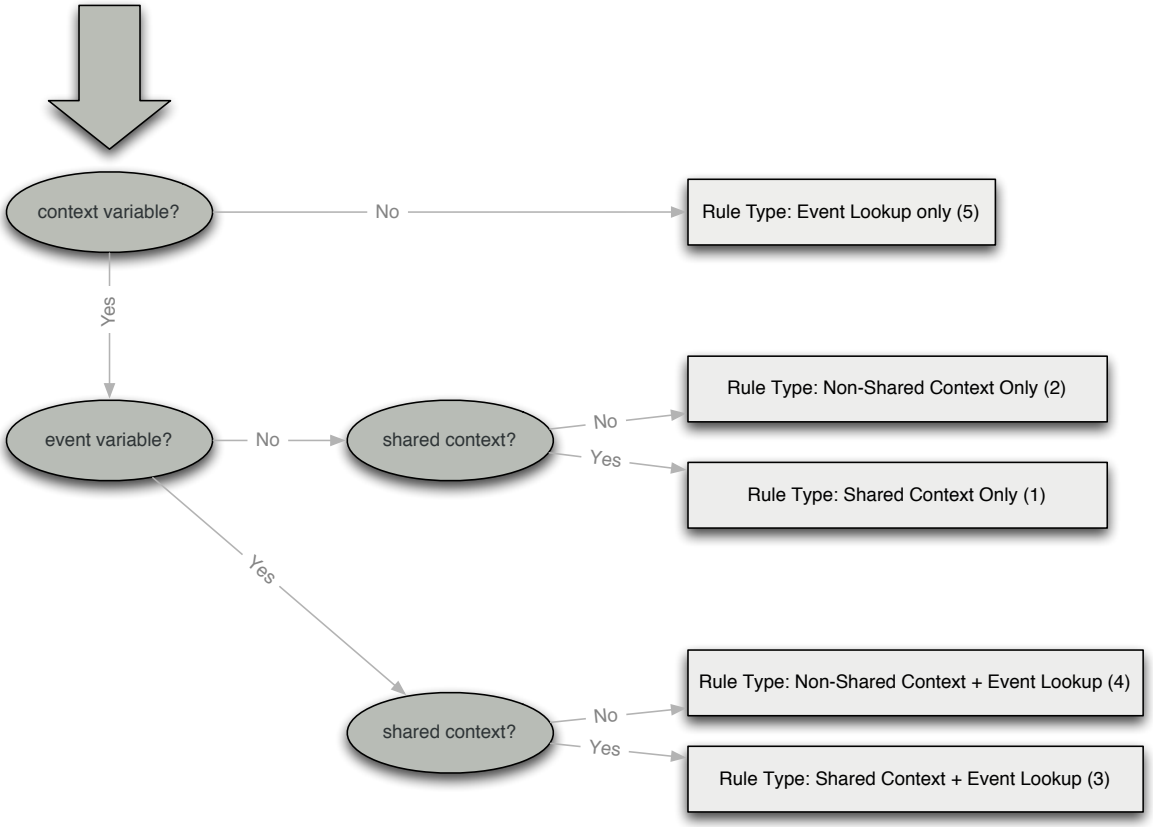

Figure 4.1: Possible Rule Types

variables inside a rule is shared with another. Using these three driving criteria we can identify the following five rule types (see figure 4.1).

Non-Shared Rule Types Rules of this type either do not contain any context variables (5), or contain context variables that are not directly shared with other rules in their respective condition lists — (4) or (2). These type of rules are of minor interest — from a rule engine perspective — as their RHS neither read nor write to shared context variables. Having a rule base that solely consists of non-shared rule types makes a rule engine unnecessary, as the context variables alone serve as unique discriminators to eliminate unrelated rules and activate the correct ones.

Shared Rule Types These type of rules contain context variables that are also shared among other rules — (1) or (3). All context variables that occur in a rule of this type automatically classify the other participating rules to be a shared rule type. These represent the majority of the rules as these require a rule engine to be in place for efficient rule activation.

Sharedness Factor

Knowing that a context variable is shared among different rules is not enough. In order to understand the degree of sharedness for a context variable, we need to define the *sharedness factor* of a context variable: $\frac{c_i}{N}$, where c_i = specific context variable and N = the total

number of rules within the inference engine. This *sharedness factor* is one of the driving input for the benchmark generator (see section 4.2.1), as we can control how far the average context variable is shared among the rule base.

4.1.2 Optimization Types

The different optimization types we will introduce in this subsection requires the understanding of the different conditions that are generated by the DSL compiler. We have touched previously on the fact, that both the event, as well as the context variables generate conditions. Here, we will explain in detail which conditions the subcomponents of the two types of conditions are composed of.

Event Name Condition & Event Argument Conditions

The event conditions are subdivided into the *event name condition* and *event argument conditions*. An event name X is represented as the singular condition '`condition "curr_evt" "name" (VVal $ ValString "X") []`'. An event argument Y is represented as the singular condition '`condition "curr_evt" "Y" (VVar $ Var "event_Y") []`'. Each event can contain an arbitrary amount of arguments increasing the amount of *event argument conditions* accordingly. All event conditions are identified with the *id* "curr_evt", making event conditions a set of dynamic WMEs. They are always modified — removed and added — every time a new event is emitted. The *event name condition's* attribute is statically set to be "name". Each event argument Y has the corresponding condition *attribute* with the same name, and a variable is assigned — in the form *event_Y* — so the correct value is retrieved during variable lookups. So for example, the event argument "instance" is identified by the *id* "curr_evt", the *attribute* "instance" and the *value* "?event_instance", the "?" denoting a variable.

Context Variable Conditions

A context variable X has a set of three conditions. The first condition represents the context type: '`condition "?ctx_X_sym" "type" (VVal $ ValString "context") []`'. The second condition represents the name of the context variable: '`condition "?cte_X_sym" "name" (VVal $ ValString "X") []`'. The last condition represents the value the context variable holds and is thus the only condition with a variable: '`condition "?ctx_X_sym" "value" (VVar $ Var "?ctx_X_value") []`'.

D - default RETE conditions structure

This is the default unoptimized condition structure, which is structured in the following way: `[ (event name), (event arg 1), ..., (event arg N), (context var 1), ..., (context var N) ]`. It starts with the *event name condition* at the top with subsequent *event argument conditions* and any number of *context variable conditions*.

C - ContextTop

The idea of ContextTop is to move the *event conditions* to the bottom of the condition list. This behavior follows the heuristic of positioning conditions affected by dynamic

WMEs directly to the bottom, which results in minimizing the evaluation cost. It is structured thusly: `[ (context var 1), ..., (context var N), (event name), (event arg 1), ..., (event arg N) ]`

E - EventArgsBeforeName

The idea of EventArgsBeforeName follows its name, in that the *event argument conditions* are positioned *before* the *event name condition*. We know by the heuristic of *place restrictive conditions first*, that the event conditions are prime candidates as restrictive conditions. We do not yet know, which of the event condition parts are more restrictive. With EventArgsBeforeName we try to see whether the *event argument conditions* are the more restrictive part of the conditions. The structure for this optimization type follows: `[ (event arg 1) , ... , (event arg N) , (event name) , (context var 1) , ... , (context var N) ]`.

EC - EventArgsBeforeNameContextTop

This optimization type combines ContextTop with EventArgsBeforeNameContextTop. It puts both the *event conditions* to the bottom of the condition list and also positions the *event argument conditions* before the *event name condition*. This idea hooks into the dynamic WME heuristic, just as ContextTop, with the difference that we identify the *event name condition* part to be more heavily affected by the dynamism of the associated WMEs. Its structure is the following: `[ (context var 1) , ... , (context var N) , (event arg 1) , ... , (event arg N) , (event name) ]`.

I - inverse event (WME) addition

The order of WME addition also affects the order of beta network evaluation. Whereas the default approach to event addition is by submitting the event name WME first, with subsequent arguments last.

```
[ (event name), (event arg 1), ..., (event arg N) ]
```

This optimization type swaps the two WMEs: name and arguments. By doing so we are hoping to minimize the beta network evaluation cost.

```
[ (event arg 1), ..., (event arg N), (event name) ]
```

L - lean context conditions

Recall that there are three conditions for context variables. One for typing the object (refer to section 2.5.1), one for naming and one for storing the value of the context variable. It turns out that in our case we can remove the typing information and thus remove a condition for each context variable. This is made possible because we do not need arbitrary type discrimination, as we only need to seperate events from context variables. Thus the following conditions are sufficient to represent a context variable.

```
[ condition "?ctx_X_sym" "name"  (VVal $ ValString "X")     []
, condition "?ctx_X_sym" "value" (VVar $ Var "ctx_X_value") []
]
```

Combining Optimization Types

Having explained the possible optimization types — each resulting in differing condition orderings — we can build the following 16 rule condition and WME addition combinations. The structure of the combinations follow the regular expression `(D|L)(E|C|EC)(?I)`. Whereas the first group determines the overall structure (either `D` or `L`), the second part controls the location of both the event and context variable conditions via `E`, `C` or `EC`. The last group of the expression decides whether inverse WME addition for events `I` should be employed. We will employ all these optimization combinations within the benchmark generator described in the next section.

D Default Rule Conditions

DC Default Rule Conditions + ContextTop

DE Default Rule Conditions + EventArgsBeforeName

DEC Default Rule Conditions + EventArgsBeforeNameContextTop

DI Default Rule Conditions + Inverse Event Addition

DCI Default Rule Conditions + ContextTop + Inverse Event Addition

DEI Default Rule Conditions + EventArgsBeforeName + Inverse Event Addition

DECI Default Rule Conditions + EventArgsBeforeNameContextTop + Inverse Event Addition

L Lean Rule Conditions

LC Lean Rule Conditions + ContextTop

LE Lean Rule Conditions + EventArgsBeforeName

LEC Lean Rule Conditions + EventArgsBeforeNameContextTop

LI Lean Rule Conditions + Inverse Event Addition

LCI Lean Rule Conditions + ContextTop + Inverse Event Addition

LEI Lean Rule Conditions + EventArgsBeforeName + Inverse Event Addition

LECI Lean Rule Conditions + EventArgsBeforeNameContextTop + Inverse Event Addition

4.2 Benchmarking Methodology

We have identified the different optimization combinations that affect the condition ordering within rule definitions. To find out which combination is the most efficient — in respect to the matching speed — we will follow a specific methodology.

4.2.1 Benchmark Generator

The methodology we're pursuing starts with the benchmark generator. Its job is to create a random yet consistent set of rules and event streams to perform benchmarking according to specific inputs. Beside the elementary goal of generating the specified number of events and rules, it solves another concern. As discussed in section 4.1.1 there are two rule types which we need to randomly allocate among the generated rules. This allocation has to follow the correct distribution, as having too many non-shared rule types would skew the matching speed results. Recall that rules of the shared rule type highly affect the matching speed compared to its counter part. Together with a rule sharedness factor as input — to control the amount of context variable sharing among the shared rules — the generator requires the exact percentages to allocate all the different rule types.

To summarize, the benchmark generator accepts the following inputs:

# of events to create the set of events.

# of rules to create the set of rules.

% of rule type (1) the percentage of rules that have no events (on the LHS) but shared context variables.

% of rule type (2) the percentage of rules that have no events (on the LHS) and only non-shared context variables.

% of rule type (3) the percentage of rules that have events (on the LHS) and shared context variables.

% of rule type (4) the percentage of rules that have events (on the LHS) and only non-shared context variables.

% or rule type (5) the percentage of rules that have only events (on the LHS).

Max. rule sharedness factor the maximal sharedness factor that is allowed per rule. 0 denotes non-sharing of context variables, whereas 1 signals sharing of context variables with all other rules.

The generation process follows this algorithm:

1. Randomly allocate rule types according to the input. In case the input does not specify 100% rule type allocation (e.g. $< 100\%$) we randomly allocate rule types. It is best to supply 100% rule types to create effective rule sets. For example, supplying 80% for rule type (3) and 20% for rule type (1) will consistently create rules of the shared rule type in the specified manner.

2. Create events and allocate them among those rules with the relevant rule types — (5), (3) and (4).

3. Create non-shared context variables and allocate them among rules with rule types (2) and (4).

4. Create shared context variables and allocate them among rules with rule types (1) and (3). The supplied maximal rule sharedness factor is used to control the number of context variables each rule holds, and how often a context variable is shared among the rules.

5. Build templates for both LHS and RHS from which we finally generate the conditions. The LHS template handles various arbitrary tests (see section 3.3.4) for each condition, whereas the RHS template is responsible for context variable manipulation. For this we limit ourselves to the list type (see section 3.3.3) with the associated manipulation logic: *list construction*, *list length lookups*, *list slicing* and general variable assignment. These random generation of arbitrary RHS action logic is made easier due to the custom byte code *stack-based interpreter* (see section 3.4.2).

6. Build the event stream according to input (# of events). These are random events chosen from the pool of possible events, and associated random event arguments.

7. Instantiate all rules for all 16 optimization combinations and run the benchmark by feeding the events from front to back to each optimization type instance in turn. Each instance is executed 100 times after which the mean runtime, as well as the standard deviation is calculated.

Benchmark Generator Report

A final report is generated which includes the following information:

- the exact time of benchmark execution
- all mean and standard deviation values for each optimisation type
- all context variables and the list of rules holding the variables
- general data (total # of variables, # of events, # of context variables, ...)
- the full input data for the benchmark generator
- the effective sharedness factor for each rule
- the detailed conditions (and its ordering) for its LHS, as well as the full RHS actions for each rule in all its optimization type instances.

An example report can be seen below, with some sections omitted for brevity. These reports are on the one hand used for creating the charts and analysis in the next section, and on the other hand critical for reproducing data.

```
Filename: genbench-report-2012-09-10-19:37:18.txt

Name,Mean,MeanLB,MeanUB,Stddev,StddevLB,StddevUB
"default rules / 10:10:0.3",0.16092064574435347,0.1574999327965081,...
"default rules (inv. event add) / 10:10:0.3",0.16087682440951462,...
"default rules + context top / 10:10:0.3",0.16042264416888355,...
"default rules + context top (inv. event add) / 10:10:0.3",0.15982840493395917,...
"default rules + event args before name / 10:10:0.3",0.1430207605667413,...
"default rules + event args before name (inv. event add) / 10:10:0.3",0.142380...
"default rules + event args before name + context top / 10:10:0.3",0.166771313...
... (omitted for brevity)

======== General Data ========
```

```
# of Variables (total): 14
# of Event Variables: 8
# of non-shared Context Variables: 0
# of shared Context Variables: 6

======== Stats: Shared Context Variables  ========
cdm: 2/10 -> 20.0% | [3,10]
hti: 2/10 -> 20.0% | [8,10]
lpj: 2/10 -> 20.0% | [1,9]
lqp: 2/10 -> 20.0% | [2,5]
rjp: 2/10 -> 20.0% | [4,6]
tln: 2/10 -> 20.0% | [7,3]

======== Stats: Rules Sharedness  ========
Rule #1 (EventSharedContext) ["lpj"] effective sharedness: 20.0%
Rule #2 (EventSharedContext) ["lqp"] effective sharedness: 20.0%
Rule #3 (EventSharedContext) ["tln","cdm"] effective sharedness: 40.0%
Rule #4 (EventSharedContext) ["rjp"] effective sharedness: 20.0%
Rule #5 (EventSharedContext) ["lqp"] effective sharedness: 20.0%
Rule #6 (EventSharedContext) ["rjp"] effective sharedness: 20.0%
Rule #7 (EventSharedContext) ["tln"] effective sharedness: 20.0%
Rule #8 (EventSharedContext) ["hti"] effective sharedness: 20.0%
Rule #9 (NoEventSharedContext) ["lpj"] effective sharedness: 20.0%
Rule #10 (NoEventSharedContext) ["hti","cdm"] effective sharedness: 40.0%

======== Config Data ========
Options { opt_rules = 10
        , opt_events = 10
        , opt_shared_context_max = 0.3
        , opt_shared_context_per_rule = 2
        , opt_rule_type_event_lookup_only = 0.0
        , opt_rule_type_event_shared_context = 0.8
        , opt_rule_type_event_non_shared_context = 0.0
        , opt_rule_type_no_event_shared_context = 0.2
        , opt_rule_type_no_event_non_shared_context = 0.0
        , opt_random_rule_types = False
        , opt_bruteforce_only = False
        }

======== Detailed Rules Data ========

Name: Rule #1 (DefaultOptimisation)
Salience: 0
Conditions:
     1: Condition curr_evt name ValString "mmp" []
     2: Condition curr_evt instance ?event_instance []
     3: Condition curr_evt activity ?event_activity []
     4: Condition ?context_lpj_sym type ValString "context" []
```

```
     5: Condition ?context_lpj_sym name ValString "lpj" []
     6: Condition ?context_lpj_sym value ?context_lpj_value []
Actions:
     1: Call (EventLookup "instance")
     2: Call (EventLookup "activity")
     3: Call (ListConstruct 2)
     4: Call (ContextLookup "lpj")
     5: Call ListAppend
Rule Type: EventSharedContext
Assigned Variables:
    1: ContextVariable "lpj"
    2: EventVariable "mmp"
-----

...
(rest omitted, which shows all rule definitions according to their optimization type)
...
```

4.3 Benchmark Results

We follow the methodology explained in section 4.2 to answer these questions:

1. How does increasing the number of events affect the matching speed?

2. How does increasing the number of rules affect the matching speed?

3. How does increasing the rule sharedness factor affect the matching speed?

By limiting the number of variables we hope to contain the effects of the overall changing RETE matching speed, and discover whether any observed speedups are statistically significant or merely accidental. In this respect table 4.1 shows all speedups for all optimization combination for the different scaling settings.

The optimization type "D" is used as the baseline benchmark in order to calculate the speedup. This is accomplished by subtracting the mean runtime of "D" from all other runtimes and using that result as the relative percentage to the baseline. The best average speedups can be observed within the "LE" and "LEI" optimization types showing a 40% speedup across all three scaling settings.

The reduced number of total conditions is responsible for the majority of the speedup effect. Nearly 30% alone is accomplished by the fact that one less condition is used for each context variable — as realized by the "L" optimization. The additional speedup average of 10% is accomplished due to "E": the moving of event arguments before the event name condition. This confirms that event argument conditions are more restrictive than the event name condition counterpart, also answering the question of optimal condition ordering for the process execution domain: [(event arg 1), ..., (event arg 2), (event name), (context name 1), (context value 1), ..., (context value N), (context value N)].

Not every optimization type contribute to the speedup. There are also some that are ineffective optimizations. Consider the "E" optimizations ("DE", "LE", "LEC", "LEC"), which reverse the event conditions by having the event name preceding the event arguments. Even though the WME submission of "I" is aligned with the event condition structure, we do

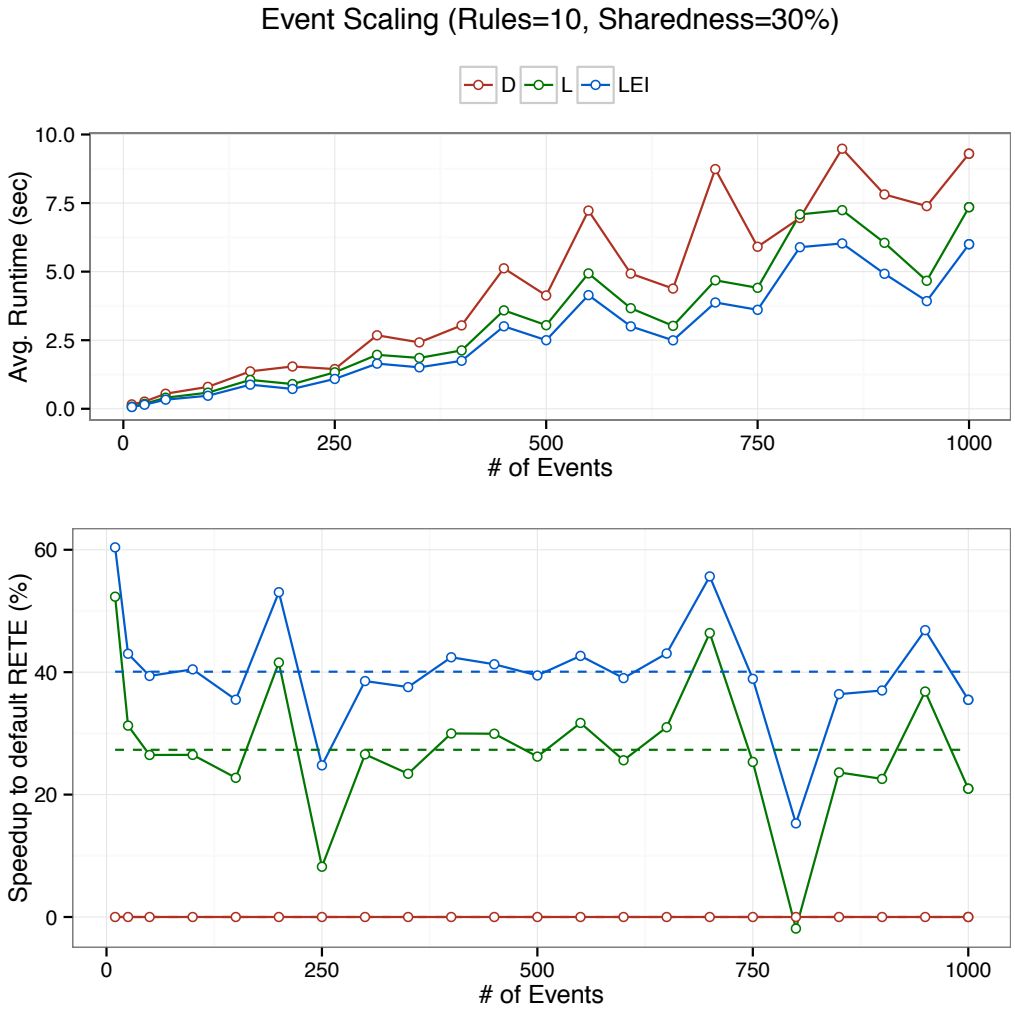

Figure 4.2: Event scaling with fixed 10 rules and sharedness factor of 30%

not see any considerable speedup differences among the "I" and non-"I" optimizations (e.g. "LE" vs "LEI"). Another ineffective optimization type is "C" (alongside "EC") moving the event conditions to the bottom — following the dynamic WME theory. We conclude that dynamic WMEs are secondary for the overall matching speed, and the higher priority lies in the restrictive property of events leading to faster evaluations for the beta network.

Figure 4.2, 4.3 and 4.4 show the detailed runtime and speedup data points with the respective scaling settings. They display the same concluding data with a finer granularity. Expectedly, the runtimes scale linearly with increasing control variables — # of events, # of rules and rule sharedness factor respectively. Small deviations — ups and downs — can be seen across data point instances, as each instance represents randomly generated benchmark data following the methodology mentioned in section 4.2. The constant speedup is confirmed as its calculation is performed within each instance for all the optimization combinations.

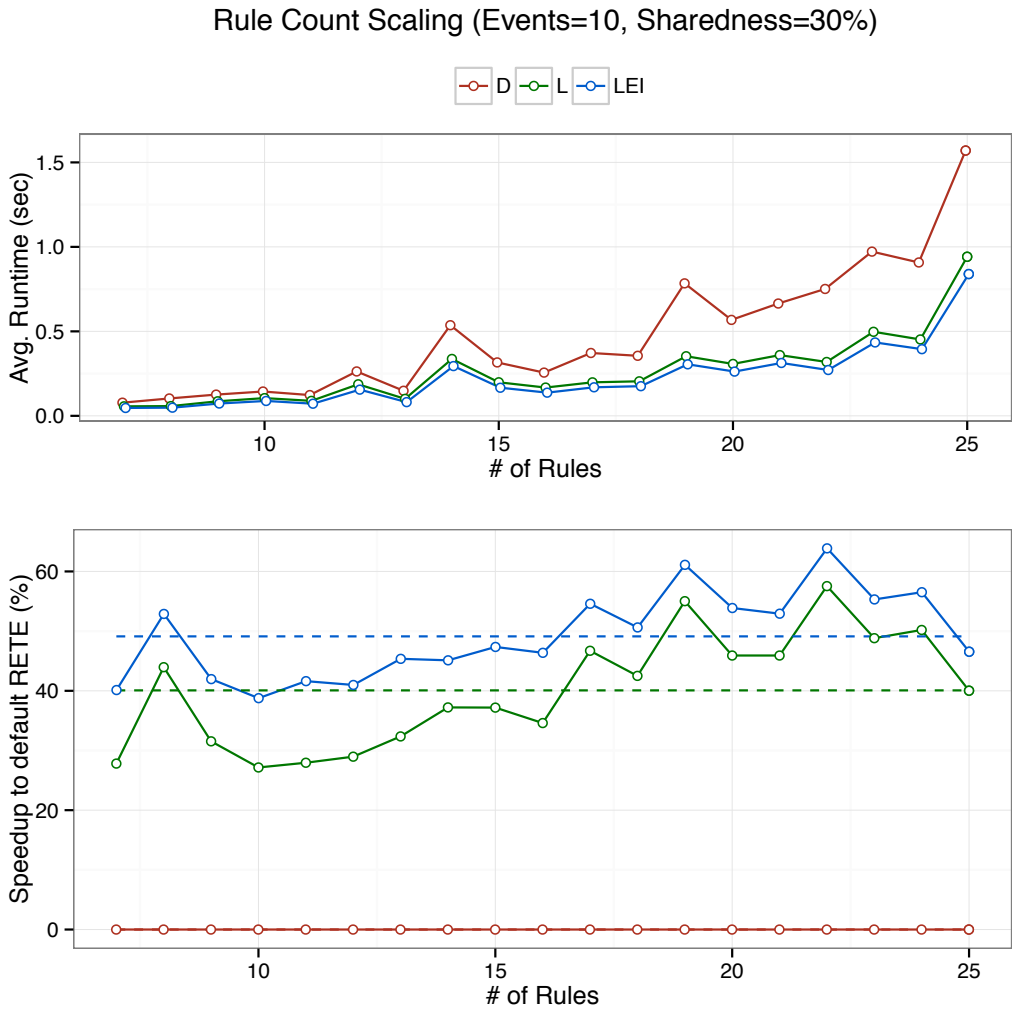

Figure 4.3: Rule scaling with fixed 10 events and sharedness factor of 30%

Optimization Type	Event Scaling	Rule Scaling	Sharedness Scaling
D	0%	0%	0%
DI	0.48%	0.19%	0.10%
DC	15.53%	-26.65%	-14.81%
DCI	15.75%	-26.79%	-14.50%
DE	13.75%	10.51%	8.05%
DEI	13.53%	10.50%	8.08%
DEC	9.69%	-30.00%	-16.65%
DECI	9.11%	-30.42%	-14.43%
L	27.31%	40.07%	31.91%
LI	27.63%	40.17%	31.98%
LC	27.36%	40.10%	31.72%
LCI	27.64%	40.22%	31.79%
LE	**40.22%**	**49.19%**	40.05%
LEI	40.07%	49.12%	**40.20%**
LEC	29.27%	34.29%	28.16%
LECI	28.92%	34.13%	28.01%

Table 4.1: Average Speedups compared to default rules in %

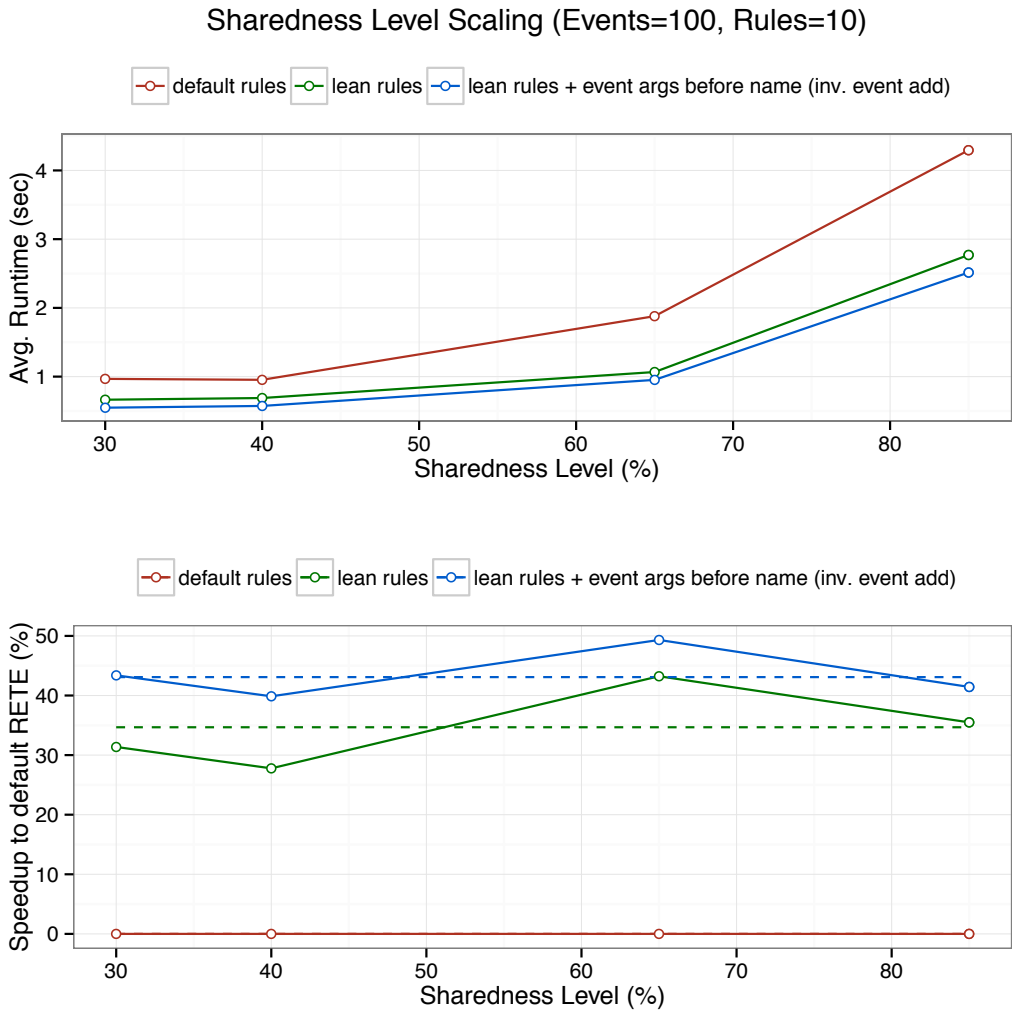

Figure 4.4: Sharedness scaling with fixed 100 events and 10 rules

4.4 Bruteforce Pattern Matching vs unoptimized RETE

For the comparison between the unoptimized RETE against the brute force forward chaining algorithm (see section 2.4.9) we have chosen to limit the benchmark to a modest size, since the runtime of the brute force approach has shown an exponential increase in rule matching time (see Fig. 4.5). The parameters we have chosen are (1) 10 fixed rules (2) 30% sharedness factor and (3) event scaling (starting from 1 up to 1000). Instead of implementing the brute force algorithm we have decided to adapt the RETE core algorithm to support a special flag: `bruteforce=True` within the configuration file. We thus simulate a brute force approach by reducing all caching effects of RETE proper by rebuilding the whole network from scratch for every WME we're adding. This includes (1) *adding* the rules beforehand and (2) *removing* the created production nodes afterwards. The RETE rule addition logic handles the case of matching already added WMEs to the Fact store. In this way the RETE network performs unnecessary left- as well as right-activations, thus losing any caching benefits inherently designed into the RETE network. We argue that this is a reasonable approximation of a brute-force pattern matching algorithm. We decided on this simulation of the brute force approach in order to avoid comparing REMAR vs W2 which would have been an *apples vs oranges* comparison since they are implemented in different programming languages — exhibiting different runtime characteristics.

4.5 Summary

We introduced the means to further optimize the RETE algorithm for the process execution context based on the idea of *conjunct ordering*.

We further described a *benchmark generator* and benchmarked various scaling effects on all three major dimensions: *# of events*, *# of rules*, *sharedness factor* and determined that the optimizations *LE* and *LEI* lead the scoreboard.

For a vanilla RETE algorithm, an optimized RETE algorithm, and a brute force algorithm we answered the following questions:

- How does increasing the number of events affect the matching speed?
- How does increasing the number of rules affect the matching speed?
- How does increasing the rule sharedness factor affect the matching speed?

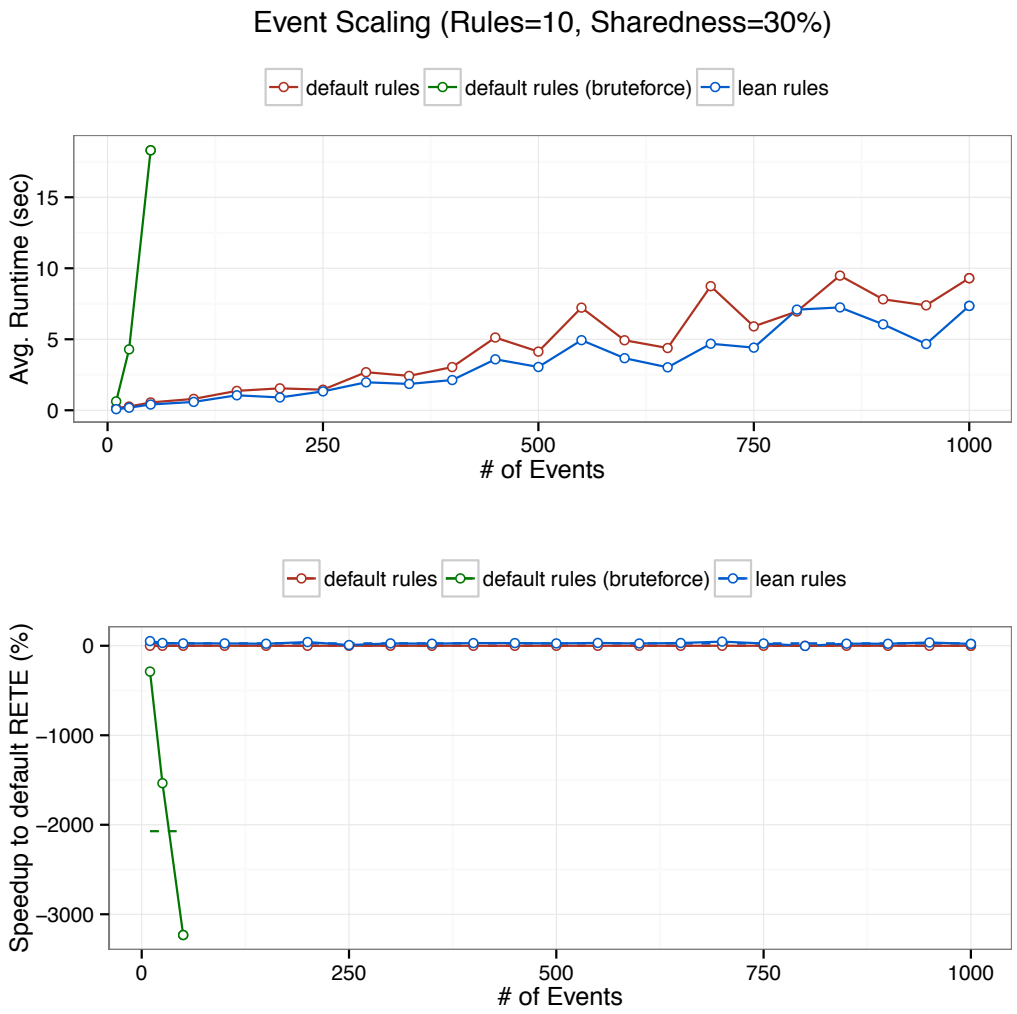

Figure 4.5: Event scaling with fixed 10 rules and sharedness factor of 30% (including brute force method)

Chapter 5

Conclusion

5.1 Summary

We implemented Workflow Watson (W2), a RESTful web service that consists of (1) a RETE-based inference engine and (2) a DSL tailored for a specific domain of process execution: synchronization.

W2 is a plug-in-replacement for REMAR [9], realizing an order-of-magnitude speedup by using the RETE algorithm. As REMAR it is implemented as an external subscriber for the CPEE [11]. Yet, any process execution engine that exposes an event stream can potentially utilize W2.

Even though the implementation of this thesis focuses on the synchronization domain, related domains can benefit from this approach, namely compliance checking and security rules. In fact, any number of non-workflow specific domains can benefit from the model proposed by W2, as any field where one follows some kind of rules can be a potential use case: tax systems and stock trading systems come to mind. Adapting W2 to a new domain is accomplished by modifying the DSL grammar and specifying domain-relevant actions. The existing DSL compiler and *stack-based interpreter* can be reused to allow a safe sandbox for restricted action execution. What these actions are is domain-dependent and can be extended.

Benchmarking a naive *unify*-based forward chaining algorithm against the RETE-based pattern matching algorithm shows clear results. The exponential slowdown due to the brute force method is unpractical for real applications. The caching nature of the RETE algorithm is required to keep up the linear run time. Although once maximum memory capacity is reached such caching facilities fail, and the algorithm approaches its limit.

Specific to the process execution domain, we could further optimize the RETE-based rule matching process by using a correct condition ordering scheme and event posting protocol (LEI) (see section 4.1.2). Doing so we could further attain an average speedup of 40% compared to a default unoptimized RETE implementation. These results are obtained by using a *benchmark generator* to observe various scaling effects on three major dimensions: *# of events*, *# of rules*, and *sharedness factor*. We conclude that the speedup is achieved on all dimensions. Domain experts working with the W2 DSL benefit from the optimization, which he/she is not required to perform manually as it is part of the DSL compilation process.

5.2 Future Work

Based on the concepts of external subscribers introduced in section 1.1.1, it becomes clear that in order to implement consistency checking, we do not need additional code in W2, but can rely on an additional external subscriber that consumes the events of W2 to realize it.

W2 itself could be improved towards multi-core based parallel pattern matching and rule activation. Our implementation based on Haskell puts us in an ideal position to experiment with concurrency and parallelism. The language's *functional* and *pure* nature paves the way in this regard. The *island-based* data structure we implemented can be extended and benchmarked in a multi-core setting.

Further more we could implement a distributed rule base for choreography settings. There are two ways to manage a rule base in such a setting: (1) centralized or (2) decentralized. The former places an unfair responsibility to the entity maintaining the central rule base, as it is his sole task to keep the service running requiring plenty of resources to accomplish. Keeping the service available without being a single point of failure is an enormous task. For a decentralized approach, there is the problem of providing a data store that is distributable and available — ideally at all times — among the partner organizations. Further, synchronization is an issue with disparate partners all able to modify the decentralized rule base concurrently. The problem of synchronization in a centralized approach would be non-existent.

Bibliography

[1] Leach, D.K.: The Iron Law of What Again? Conceptualizing Oligarchy Across Organizational Forms. Sociological Theory **23**(3) (September 2005) 312–337

[2] Scheer, A.W.W.: Aris-Business Process Frameworks. 2nd edn. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1998)

[3] Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)

[4] Russell, N.C.: Foundations of process-aware information systems. PhD thesis, Queensland University of Technology (2007)

[5] Womack, J.P., Jones, D.T., Roos, D.: The Machine That Changed the World : The Story of Lean Production. Harper Perennial (1991)

[6] White, S.: Introduction to bpmn. Technical report (2004)

[7] Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: Declare: Full support for loosely-structured processes. In: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference. EDOC '07, Washington, DC, USA, IEEE Computer Society (2007) 287–

[8] Bae, J., Bae, H., Kang, S.H., Kim, Y.: Automatic control of workflow processes using eca rules. IEEE Trans. on Knowl. and Data Eng. **16**(8) (August 2004) 1010–1023

[9] Mangler, J., Rinderle-Ma, S.: Rule-based synchronization of process activities. In: Proceedings of the 2011 IEEE 13th Conference on Commerce and Enterprise Computing. CEC '11, Washington, DC, USA, IEEE Computer Society (2011) 121–128

[10] Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1) (July 2003) 5–51

[11] Stuermer, G., Mangler, J., Schikuta, E.: Building a modular service oriented workflow engine. In: IEEE International Conference on Service-Oriented Computing and Applications (SOCA'09), IEEE (December 2009)

[12] Mangler, J., Witzany, C., Schikuta, E.: Quo vadis interface definition languages? towards an interface definition language for restful services. In: IEEE International Conference on Service-Oriented Computing and Applications (SOCA'09), IEEE (December 2009)

[13] Mangler, J., Beran, P.P., Schikuta, E.: On the origin of services - using riddl for description, evolution and composition of restful services. In: The 10th IEEE?ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE (May 2010)

[14] Lu, R., Sadiq, S., Governatori, G.: Compliance aware business process design. In: Proceedings of the 2007 international conference on Business process management. BPM'07, Berlin, Heidelberg, Springer-Verlag (2008) 120–131

[15] Abdullah, N.S., Sadiq, S., Indulska, M.: Emerging challenges in information systems research for regulatory compliance management. In: Proceedings of the 22nd international conference on Advanced information systems engineering. CAiSE'10, Berlin, Heidelberg, Springer-Verlag (2010) 251–265

[16] Kharbili, M.E., Stein, S., Markovic, I., Pulvermüller, E.: Towards a framework for semantic business process compliance management (2007)

[17] Sadiq, S., Governatori, G., Naimiri, K.: Modelling control objectives for business process compliance. In: In Proc. 5th International Conference on Business Process Management. (2007) 24–28

[18] Lanz, A., Weber, B., Reichert, M.: Workflow time patterns for process-aware information systems. In: Proceedings Enterprise, Business-Process, and Information Systems Modelling: 11th International Workshop BPMDS and 15th International Conference EMMSAD at CAiSE 2010. Number 50 in LNBIP, Springer (June 2010) 94–107

[19] van der Aalst, W.M.P., de Beer, H.T., van Dongen, B.F.: Process mining and verification of properties: an approach based on temporal logic. In: Proceedings of the 2005 Confederated international conference on On the Move to Meaningful Internet Systems - Volume Part I. OTM'05, Berlin, Heidelberg, Springer-Verlag (2005) 130–147

[20] Awad, A., Decker, G., Weske, M.: Efficient compliance checking using bpmn-q and temporal logic. In: Proceedings of the 6th International Conference on Business Process Management. BPM '08, Berlin, Heidelberg, Springer-Verlag (2008) 326–341

[21] Maggi, F.M., Montali, M., Westergaard, M., Van Der Aalst, W.M.P.: Monitoring business constraints with linear temporal logic: an approach based on colored automata. In: Proceedings of the 9th international conference on Business process management. BPM'11, Berlin, Heidelberg, Springer-Verlag (2011) 132–147

[22] Laue, R., Awad, A.: Visualization of business process modeling anti patterns. ECEASST **25** (2010)

[23] Verbeek, H., Buijs, J., Dongen, B., Aalst, W.: ProM 6: The Process Mining Toolkit. In Rosa, M.L., ed.: Proc. of BPM Demonstration Track 2010. Volume 615 of CEUR Workshop Proceedings. (2010) 34–39

[24] Maggi, F.M., Montali, M., van der Aalst, W.M.P.: An operational decision support framework for monitoring business constraints. In: Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering. FASE'12, Berlin, Heidelberg, Springer-Verlag (2012) 146–162

[25] Beheshti, S.M.R., Benatallah, B., Motahari-Nezhad, H.R., Sakr, S.: A query language for analyzing business processes execution. In: Proceedings of the 9th international conference on Business process management. BPM'11, Berlin, Heidelberg, Springer-Verlag (2011) 281–297

[26] Leitner, M.: Security policies in adaptive process-aware information systems: Existing approaches and challenges. In: Proceedings of the 2011 Sixth International Conference on Availability, Reliability and Security. ARES '11, Washington, DC, USA, IEEE Computer Society (2011) 686–691

[27] Reichert, M., Dadam, P.: Adept flex - supporting dynamic changes of workflows without loosing control. Journal of Intelligent Information Systems **10** (1998) 93–129

[28] Dayal, U., Hsu, M., Ladin, R.: Organizing long-running activities with triggers and transactions. In: Proceedings of the 1990 ACM SIGMOD international conference on Management of data. SIGMOD '90, New York, NY, USA, ACM (1990) 204–214

[29] Russell, S.J., Norvig, P., Candy, J.F., Malik, J.M., Edwards, D.D.: Artificial Intelligence: A Modern Approach (2nd Edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (2002)

[30] Scales, D.J.: Efficient matching algorithms for the soariopss production system. Technical report, Stanford, CA, USA (1986)

[31] Feigenbaum, E.A.: Expert systems: Principles and practice (1992)

[32] Dreyfus, S.E., Dreyfus, H.L.: A Five-Stage Model of the Mental Activities Involved in Directed Skill Acquisition. Technical report (February 1980)

[33] Brachman, R., Levesque, H.: Knowledge Representation and Reasoning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)

[34] Lindsay, R.K., Buchanan, B.G., Feigenbaum, E.A., Lederberg, J.: DENDRAL: A Case Study of the First Expert System for Scientific Hypothesis Formation. Artificial Intelligence **61** (1993) 209–261

[35] Buchanan, B.G., Shortliffe, E.H.: Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project (The Addison-Wesley series in artificial intelligence). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1984)

[36] Barker, V.E., O'Connor, D.E., Bachant, J., Soloway, E.: Expert systems for configuration at digital: Xcon and beyond. Commun. ACM **32**(3) (March 1989) 298–318

[37] Smith, R.G., Young, R.L.: The design of the dipmeter advisor system. In: Proceedings of the 1984 annual conference of the ACM on The fifth generation challenge. ACM '84, New York, NY, USA, ACM (1984) 15–23

[38] Dzierzanowski, J., Hestenes, E., Lawson, S.: The credit assistant: the second leg in the knowledge highway for american express. In: Proceedings of the fourth conference on Innovative applications of artificial intelligence. IAAI'92, AAAI Press (1992) 127–134

[39] Ferrucci, D.A.: Ibm's watson/deepqa. In: Proceedings of the 38th annual international symposium on Computer architecture. ISCA '11, New York, NY, USA, ACM (2011) –

[40] Ferrucci, D.: Build watson: an overview of deepqa for the jeopardy! challenge. In Salapura, V., Gschwind, M., Knoop, J., eds.: 19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010), Vienna, Austria, September 11-15, 2010, ACM (2010) 1–2

[41] Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. (1990) 324–341

[42] Tambe, M., Kalp, D., Rosenbloom, P.: Uni-rete: Specializing the rete match algorithm for the unique-attribute representation. Technical report (1991)

[43] Sosnowski, Z.A.: Chaining of fuzzy rules in rete network. In: Proceedings of the International Conference, 7th Fuzzy Days on Computational Intelligence, Theory and Applications, London, UK, UK, Springer-Verlag (2001) 895–903

[44] Walzer, K., Groch, M., Breddin, T.: Time to the rescue - supporting temporal reasoning in the rete algorithm for complex event processing. In: Proceedings of the 19th international conference on Database and Expert Systems Applications. DEXA '08, Berlin, Heidelberg, Springer-Verlag (2008) 635–642

[45] Schmedding, F., Sawas, N., Lausen, G.: Adapting the rete-algorithm to evaluate f-logic rules. In: Proceedings of the 2007 international conference on Advances in rule interchange and applications. RuleML'07, Berlin, Heidelberg, Springer-Verlag (2007) 166–173

[46] Doorenbos, R.B.: Production matching for large learning systems. PhD thesis, Pittsburgh, PA, USA (1995) UMI Order No. GAX95-22942.

[47] Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: A history of haskell: being lazy with class. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages. HOPL III, New York, NY, USA, ACM (2007) 12–1–12–55

[48] Kernighan, B.W., Ritchie, D.M.: The C Programming Language. 2nd edn. Prentice Hall Professional Technical Reference (1988)

[49] Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. PPoPP '05, New York, NY, USA, ACM (2005) 48–60

[50] Marlow, S.: Parallel and concurrent programming in Haskell. In Zsók, V., Horváth, Z., Plasmeijer, R., eds.: CEFP 2011. Volume 7241 of LNCS. (2012) 339–401

[51] Oram, A., Wilson, G.: Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly)). O'Reilly Media, Inc. (2007)

[52] Discolo, A., Harris, T., Marlow, S., Jones, S.P., Singh, S.: Lock free data structures using stm in haskell. In: Proceedings of the 8th international conference on Functional and Logic Programming. FLOPS'06, Berlin, Heidelberg, Springer-Verlag (2006) 65–80

[53] Sulzmann, M., Lam, E.S., Marlow, S.: Comparing the performance of concurrent linked-list implementations in haskell. SIGPLAN Not. **44**(5) (October 2009) 11–20

[54] Kanal, L., Kumar, V., Kitano, H., Suttner, C., Amaral, J.N., Ghosh, J.: Speeding up production systems: From concurrent matching to parallel rule firing (1993)

[55] Huet, G.: The zipper. J. Funct. Program. **7**(5) (September 1997) 549–554

[56] Adams, M.D.: Scrap your zippers: a generic zipper for heterogeneous types. In: Proceedings of the 6th ACM SIGPLAN workshop on Generic programming. WGP '10, New York, NY, USA, ACM (2010) 13–24

[57] : Tying the knot. `http://www.haskell.org/haskellwiki/Tying_the_Knot` Accessed: 30/06/2012.

[58] Parr, T.: Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. 1st edn. Pragmatic Bookshelf (2009)

[59] Strembeck, M., Zdun, U.: An approach for the systematic development of domain-specific languages. Software: Practice and Experience **39**(15) (10 2009)

[60] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4) (December 2005) 316–344

[61] Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: Lua—an extensible extension language. Software: Practice and Experience **26**(6) (1996) 635–652

[62] Ishida, T.: Optimizing rules in production system programs. In: In Proceedings of National Conference on Artificial Intelligence. (1988) 699–704

[63] Nayak, P., Gupta, A., Rosenbloom, P.S.: The soar papers (vol. 1). MIT Press, Cambridge, MA, USA (1993) 621–626

List of Figures

Appendix A

Abstract

A.1 English

The environment in which business processes exist are complex. Business processes are affected by many factors — external as well as internal — such as ever changing security requirements, compliance to laws and regulations, as well as scarce resources. Modeling all these changes directly into the processes itself would lead to very volatile processes — whereas the typical goal of a process is to describe the strategic business logic of a company in a simple and coherent way.

In order to stay flexible and keep the original process definitions simple, it is desirable to manage these influences decoupled from the process definition itself. In this thesis we implement *Workflow Watson* (W2), a high performance Event-Condition-Action (ECA) based rule engine specifically optimized for the case of analyzing an event stream from a process engine. The W2 model consists of (1) a RETE-based inference engine core and (2) a DSL layer for grammar and domain-specific (e.g. security, synchronization, compliance) action definition. Our contribution to (1) is a solution to the *conjunct ordering* problem for the process execution domain that improves the performance of a vanilla RETE algorithm by nearly 50%. For (2) we can adapt W2 for different domains by specifying the grammar and actions for each domain.

A.2 Deutsch

In der heutigen Zeit immer schnellerer Anpassungen an sich verändernde Bedingungen in einem globalen Wirtschaftssystem, werden auch die Geschäftsprozesse, welche z.B. der Erstellung von Produkten, oder der Interaktion mit Kunden zugrunde liegen, immer komplizierter. Geschäftsprozesse werden typischerweise von verschiedenen internen und externen Faktoren beeinflusst. Zu diesen Faktoren zählen unter anderem sich ständig ändernde Sicherheitsstandards, die Einhaltung von Gesetzen oder (firmeneigener) Vorschriften, aber auch die Lösung von Ressourcenproblemen. Diese Veränderungen direkt in den Geschäftsprozessmodellen selbst abzubilden würde im Extremfall zu sehr häufigen Anpassungen führen, und verstellt den Blick auf die eigentliche Geschäftslogik einer Firma.

Um Flexibilität und Einfachheit von Prozessmodellen zu gewährleisten, ist es erstrebenswert die internen und externen Faktoren entkoppelt von der Prozessdefinition abzubilden und zu überwachen. Der Kern dieser Arbeit ist die Implementierung von *Workflow Watson* (W2), einer hochperformanten Event-Condition-Action (ECA) basierten *Rule Engine*, welche speziell für den Fall der Analyse von *event streams* einer *Prozess Engine* optimiert ist. Das W2-Modell besteht aus (1) einer RETE-basierten *Inference Engine* und (2) einer *DSL* Ebene um anwendungsspezifische Sprachen zu definieren, welche den einfachen Umgang mit verschiedenen Anwendungsdomänen erlauben (z.B Sicherheit, *Synchronisation* von Ressourcen, oder die Überwachung von *Business Compliance*). Der wissenschaftliche Beitrag dieser Arbeit umfasst: Erstens die Lösung des *conjunct ordering*-Problem speziell *event streams* von Prozess Engines, welche im Vergleich mit einem unoptimierten RETE Algorithmus eine um 50% höhere Ausführungsgeschwindigkeit erlaubt; Zweitens wird es durch W2 möglich für verschiedene Domänen angepasste Grammatiken (DSLs) zu definieren und zu verwenden. Diese DSLs erlauben die einfache regel-basierte Lösung verschiedener Problemstellungen (siehe oben).

Appendix B

Curriculum Vitae

Persönliche Daten

Name:	Conrad Indiono
Geburtsdatum:	03.08.1984
Geburtsort:	Malang, Indonesien

Ausbildung

Universität Wien, Wien, Österreich

Okt 2010 - Jan 2013:	Masterstudium Wirtschaftsinformatik
Okt 2006 - Jul 2010:	Bachelorstudium Informatik (Schwerpunkt Wirtschaftsinformatik)

Publikationen

Data Transformation and Semantic Log Purging for Process Mining
Ly, Linh Thao and Indiono, Conrad and Mangler, Jürgen and Rinderle-Ma, Stefanie
In: International Conference on Advanced Information Systems Engineering (CaISE 2012)