Rule Engines and Production Rule Systems

Chapter 1. Introduction

1.1. Artificial Intelligence

1.1.1. A Little History

Over the last few decades ***artificial intelligence*** (AI) became an unpopular term, with the well know "AI Winter". There were large boasts from scientists and engineers looking for funding, that never lived up to expectations along with many failed projects. Thinking Machines Corporation and the 5th Generation Computer (5GP) project probably exemplify best the problems at the time.

Thinking Machines Corporation was one of the leading AI firms in 1990, it had sales of nearly $65 million. Here is quote from it's brochure:

“Some day we will build a thinking machine. It will be a truly intelligent machine. One that can see and hear and speak. A machine that will be proud of us.”

Yet 5 years later it filed for Chapter 11. inc.com has a fascinating article titled "The Rise and Fall of Thinking Machines". The article covers the growth of the industry and how a cosy relationship with Thinking Machines and DARPA over heated the market, to the point of collapse. It explains how and why commerce moved away from AI and towards more practical number crunching super computers.

The 5th Generation Computer project was a 400mill USD project in Japan to build a next generation computer. Valves was first, transistors was second, integrated circuits was third and finally microprocessors was fourth. This project spurred an "arms" race with the UK and USA, that caused much of the AI bubble. The 5GP would provide massive multi-cpu parallel processing hardware along with powerful knowledge representation and reasoning software via ***Prolog***; a type of ***expert system***. By 1992 the project was considered a failure and

cancelled. It was the largest and most visible commercial venture for Prolog, and many of the failures are pinned on the problems trying to run a logic based programming language concurrently on multi cpu hardware with effective results. Some believe that the failure of the 5GP project tainted Prolog and resigned it academia, see **"Whatever Happened to Prolog"** by John C. Dvorak.

However while research funding dried up and the term AI became less used, many green shoots where planted and continued more quietly under discipline specific names: ***cognitive systems***, ***machine learning***, ***intelligent systems***, ***knowledge representation and reasoning***. Offshoots of these then made their way into commercial systems, such as expert systems in the ***Business Rules Management System*** (BRMS) market.

Imperative, system based languages, languages such as C, C++, Java and .Net have dominated the last 20 years. Enabled by the practicality of the languages and ability to run with good performance on commodity hardware. However many believe there is renaissance undergoing in the field of AI, spurred by advances in hardware capabilities and AI research. In 2005 Heather Havenstein authored **"Spring comes to AI winter"** which outlines a case for this resurgence, which she refers to as a ***spring***. Norvig and Russel dedicate several pages to what factors allowed the industry to over come it's problems and the research that came about as a result:

“Recent years have seen a revolution in both the content and the methodology of work in artificial intelligence. It is now more common to build on existing theories than to propose brand-new ones, to base claims on rigorous theorems or hard experimental evidence rather than on intuition, and to show relevance to real-world applications rather than toy examples.” (Artificial Intelligence : A Modern Approach.)

Computer vision, ***neural networks***, ***machine learning*** and ***knowledge representation and reasoning*** (KRR) have made great strides in become practical in commercial environments. For example vision based systems can now fully map out and navigate their environments with strong recognition skills, as a result we now have self driving cars about to enter the commercial market. ***Ontological*** research, based around description logic, has provided very rich semantics to represent our world. Algorithms such as the tableaux algorithm have made it possible to effectively use those rich semantics in large complex ontologies. Early KRR systems, like Prolog in 5GP, were dogged by the limited semantic capabilities and memory restrictions on the size of those ontologies.

1.1.2. Knowledge Representation and Reasoning

In A Little History talks about AI as a broader subject and touches on Knowledge Representation and Reasoning (KRR) and also Expert Systems, I'll come back to Expert Systems later.

KRR is about how we represent our knowledge in symbolic form, i.e. how we describe something. Reasoning is about how we go about the act of thinking using this knowledge. System based languages, like Java or C+, have classification systems, called Classes, to be able to describe things, in Java we calls these things beans or instances. However those classification systems are limited to ensure computational efficiency. Over the years researchers have developed increasingly sophisticated ways to represent our world, many of you may already have heard of OWL (Web Ontology Language). Although there is always a gap between what we can be theoretically represented and what can be used computationally in practically timely manner, which is why OWL has different sub languages from Lite to Full. It is not believed that any reasoning system can support OWL Full. Although Each year algorithmic advances try and narrow that gap and improve expressiveness available to reasoning engines.

There are also many approaches to how these systems go about thinking. You may have heard of discussions comparing the merits of forward chaining, which is reactive and data driven, or backward chaining, which is passive and query driven. Many other types of reasoning techniques exists, each of which enlarges the scope of the problems we can tackle declaratively. To list just a few: imperfect reasoning (fuzzy logic, certainty factors), defeasible logic, belief systems, temporal reasoning and correlation. Don't worry if some of those words look alien to you, they aren't needed to understand Drools and are just there to give an idea of the range of scope of research topics; which is actually far more extensive than this small list and continues to grow as researches push new boundaries.

KRR is often refereed as the core of Artificial Intelligence Even when using biological approaches like neural networks, which model the brain and are more about pattern recognition than thinking, they still build on KRR theory. My first endeavours with Drools were engineering oriented, as I had no formal training or understanding of KRR. Learning KRR has allowed me to get a much wider theoretical background. Allowing me to better understand both what I've done and where I'm going, as it underpins nearly all of the theoretical side to our Drools R&D. It really is a vast and fascinating subject that will pay dividends for those that take the time learn, I know it did and still does for me. Bracham and Levesque have written a seminal piece of work, called "Knowledge Representation and Reasoning" that for anyone wanting to build strong foundations is a must read. I would also recommend the Russel and Norvig book "Artificial Intelligence, a modern approach" which also covers KRR.

1.1.3. Rule Engines and Production Rule Systems (PRS)

We've now covered a brief history of AI and learnt that the core of AI is formed around KRR. We've shown than KRR is vast and fascinating subject which forms the bulk of the theory driving Drools R&D.

The rule engine is the computer program that delivers KRR functionality to the developer. At a high level it has three components:

- Ontology
- Rules
- Data

As previous mentioned the ontology is the representation model we use for our "things". It could be a simple records or Java classes or full blown OWL based ontologies. The Rules do the reasoning and facilitate thinking. The distinction between rules and ontologies blurs a little with OWL based ontologies, who's richness is rule based.

The term rule engine is quite ambiguous in that it can be any system that uses rules, in any form, that can be applied to data to produce outcomes. This includes simple systems like form validation and dynamic expression engines. The book "How to Build a Business Rules Engine (2004)" by Malcolm Chisholm exemplifies this ambiguity. The book is actually about how to build and alter a database schema to hold validation rules. The book then shows how to generate VB code from those validation rules to validate data entry. Which while very valid, it is very different to what we talking about so far.

Drools started life as a specific type of rule engine called a production rule system (PRS) and was based around the Rete algorithm. The Rete algorithm, developed by Charles Forgey in 1979, forms the brain of a Production Rules System and is able to scale to a large number of rules and facts. A Production Rule is a two-part structure: the engine matches facts and data against Production Rules - also called Productions or just Rules - to infer conclusions which result in actions.

```
when

    <conditions>

then

    <actions>;
```

The process of matching the new or existing facts against Production Rules is called ***pattern matching***, which is performed by the ***inference engine***. Actions execute in response to changes in data, like a database trigger; we say this is a ***data driven*** approach to reasoning. The actions themselves can change data, which in turn could match against other rules causing them to fire; this is referred to asforward chaining

Drools implements and extends the Rete algorithm;. The Drools Rete implementation is called ReteOO, signifying that Drools has an enhanced and optimized implementation of the Rete algorithm for object oriented systems. Our more recent work goes well beyond Rete. Other Rete based engines also have marketing terms for their proprietary enhancements to Rete, like RetePlus and Rete III. Th e most common enhancements are covered in "Production Matching for Large Learning Systems (Rete/UL)" (1995) by Robert B. Doorenbos. Leaps used to be provided but was retired as it became unmaintained, the good news is our research is close to producing an algorithm that merges the benefits of Leaps with Rete.

The Rules are stored in the Production Memory and the facts that the Inference Engine matches against are kept in the Working Memory. Facts are asserted into the Working Memory where they may then be modified or retracted. A system with a large number of rules and facts may result in many rules being true for the same fact assertion; these rules are said to be in conflict. The Agenda manages the execution order of these conflicting rules using a Conflict Resolution strategy.

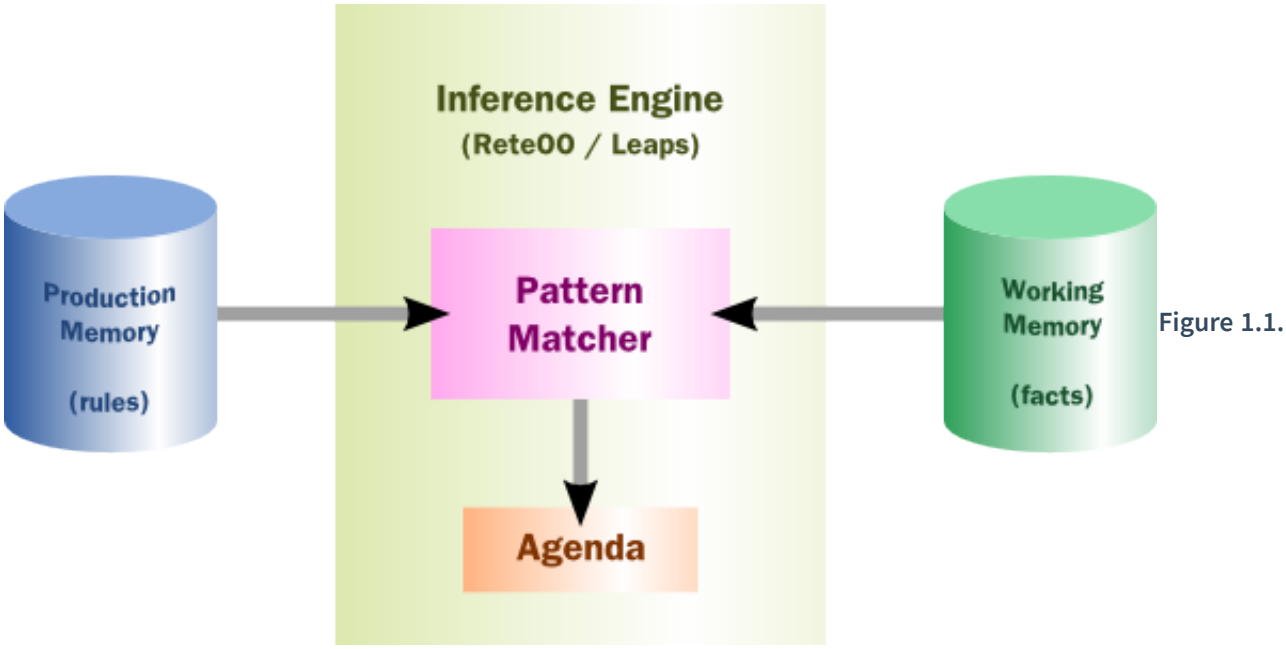


Figure 1.1. High-level View of a Production Rule System

1.1.4. Hybrid Reasoning Systems (HRS)

You may have read discussions comparing the merits of forward chaining (reactive and data driven) or backward chaining(passive query). Here is a quick explanation of these two main types of reasoning.

Forward chaining is "data-driven" and thus reactionary, with facts being asserted into working memory, which results in one or more rules being concurrently true and scheduled for execution by the Agenda. In short, we start with a fact, it propagates and we end in a conclusion.

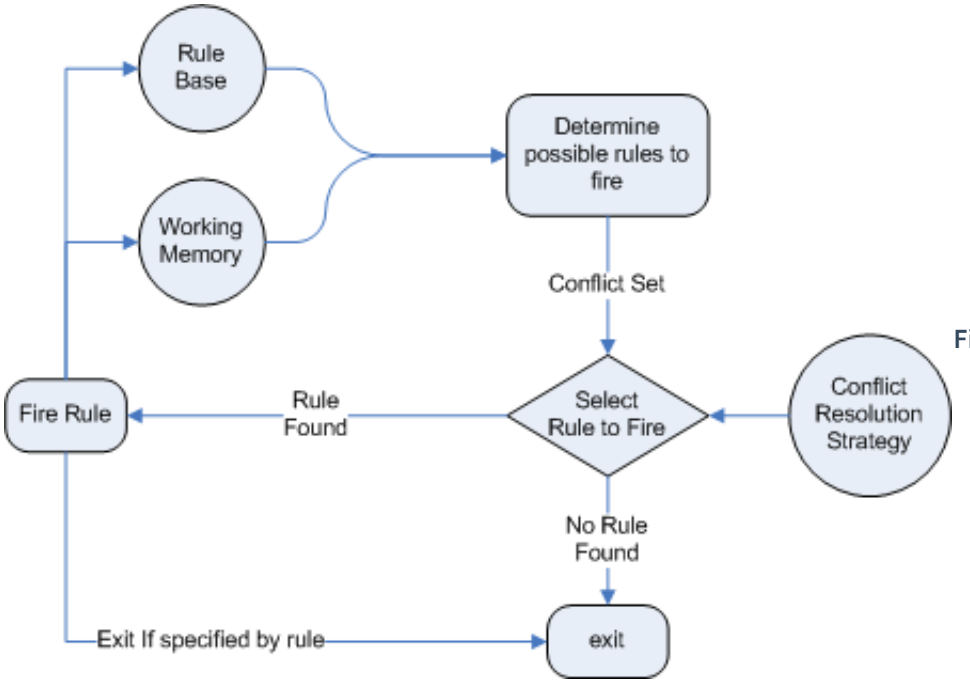


Figure 1.2. Forward Chaining

Backward chaining is "goal-driven", meaning that we start with a conclusion which the engine tries to satisfy. If it can't it then searches for conclusions that it can satisfy; these are known as subgoals, that will help satisfy some unknown part of the current goal. It continues this process until either the initial conclusion is proven or there are no more subgoals. Prolog is an example of a Backward Chaining engine. Drools can also do backward chaining, which we refer to as derivation queries.

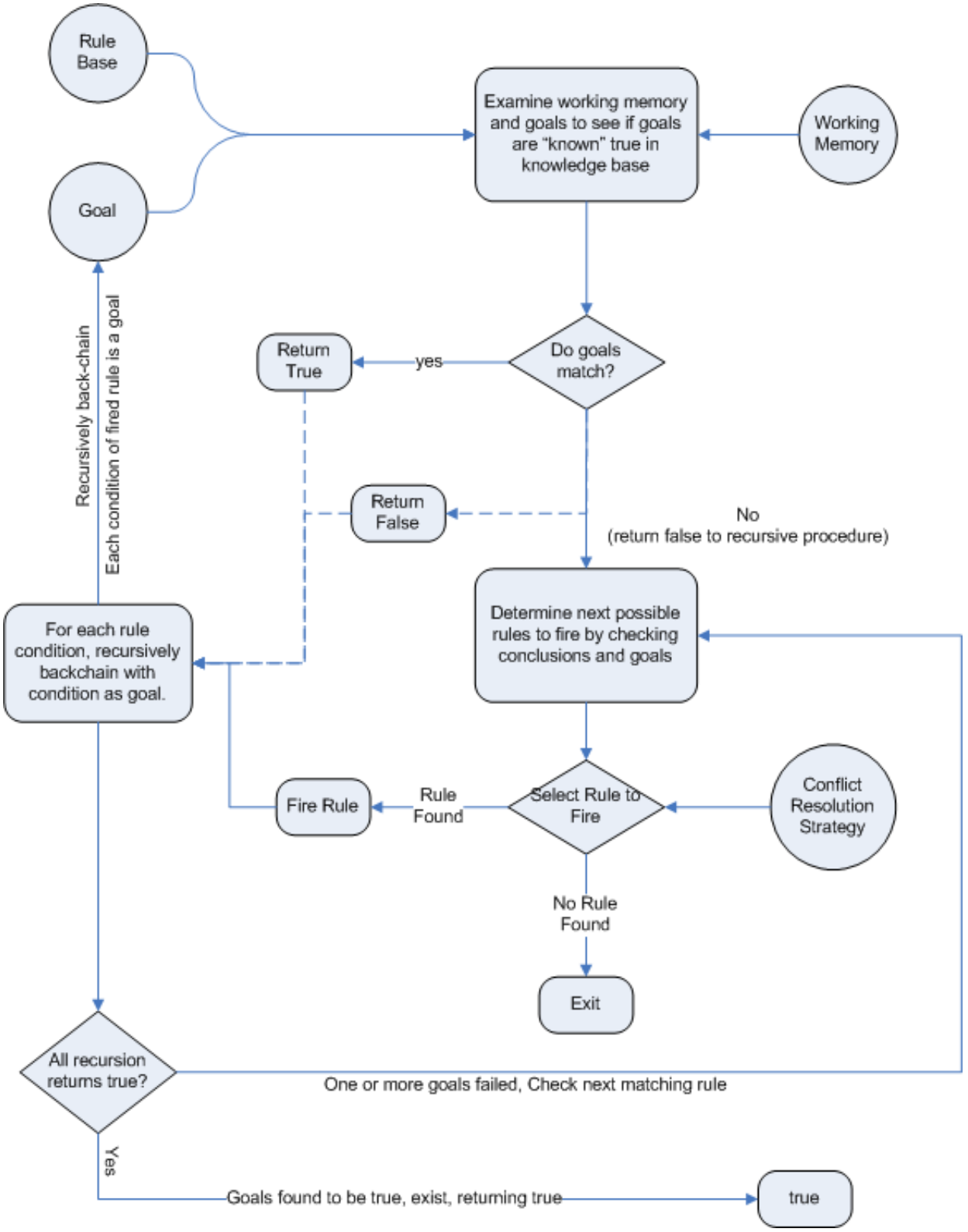


Figure 1.3. Backward Chaining

Historically you would have to make a choice between systems like OPS5 (forward) or Prolog (backward). Now many modern systems provide both types of reasoning capabilities. There are also many other types of reasoning techniques, each of which enlarges the scope of the problems we can tackle declaratively. To list just a few: imperfect reasoning (fuzzy logic, certainty factors), defeasible logic, belief systems, temporal reasoning

and correlation. Modern systems are merging these capabilities, and others not listed, to create ***hybrid reasoning systems*** (HRS).While Drools started out as a PRS, 5.x introduced Prolog style backward chaining reasoning as well as some functional programming styles. For this reason HRS is now the preferred term when referring to Drools, and what it is.Drools current provides crisp reasoning, but imperfect reasoning is almost ready. Initially this will be imperfect reasoning with fuzzy logic, later we'll add support for other types of uncertainty. Work is also under way to bring OWL based ontological reasoning, which will integrate with our ***traits*** system. We also continue to improve our functional programming capabilities.

1.1.5. Expert Systems

You will often hear the terms ***expert systems*** used to refer to ***production rule systems*** or ***Prolog*** like systems. While this is normally acceptable, it's technically wrong as these are frameworks to build expert systems with, and not actually expert systems themselves. It becomes an expert system once there is an ontological model to represent the domain and there are facilities for knowledge acquisition and explanation.

Mycin is the most famous expert system, built during the 70s. It is still heavily covered in academic literature, such as the recommended book "Expert Systems" by Peter Jackson.

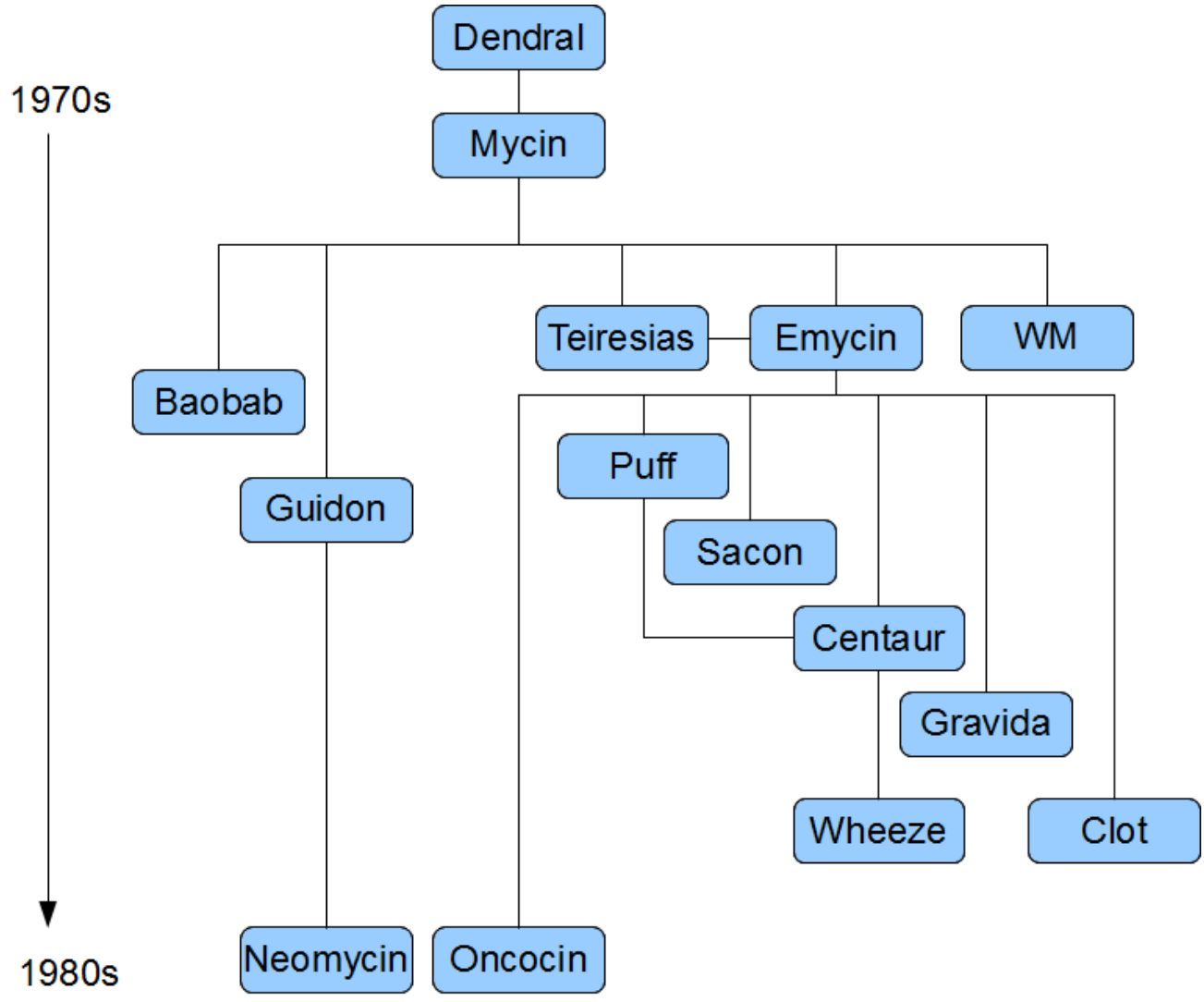


Figure 1.4. Early History of Expert Systems

1.1.6. Recommended Reading

General AI, KRR and Expert System Books

For those wanting to get a strong theoretical background in KRR and expert systems, I'd strongly recommend the following books. "Artificial Intelligence: A Modern Approach" is must have, for anyone's bookshelf.

- Introduction to Expert Systems
    - Peter Jackson
- Expert Systems: Principles and Programming
    - Joseph C. Giarratano, Gary D. Riley
- Knowledge Representation and Reasoning
    - Ronald J. Brachman, Hector J. Levesque
- Artificial Intelligence : A Modern Approach.
    - Stuart Russell and Peter Norvig

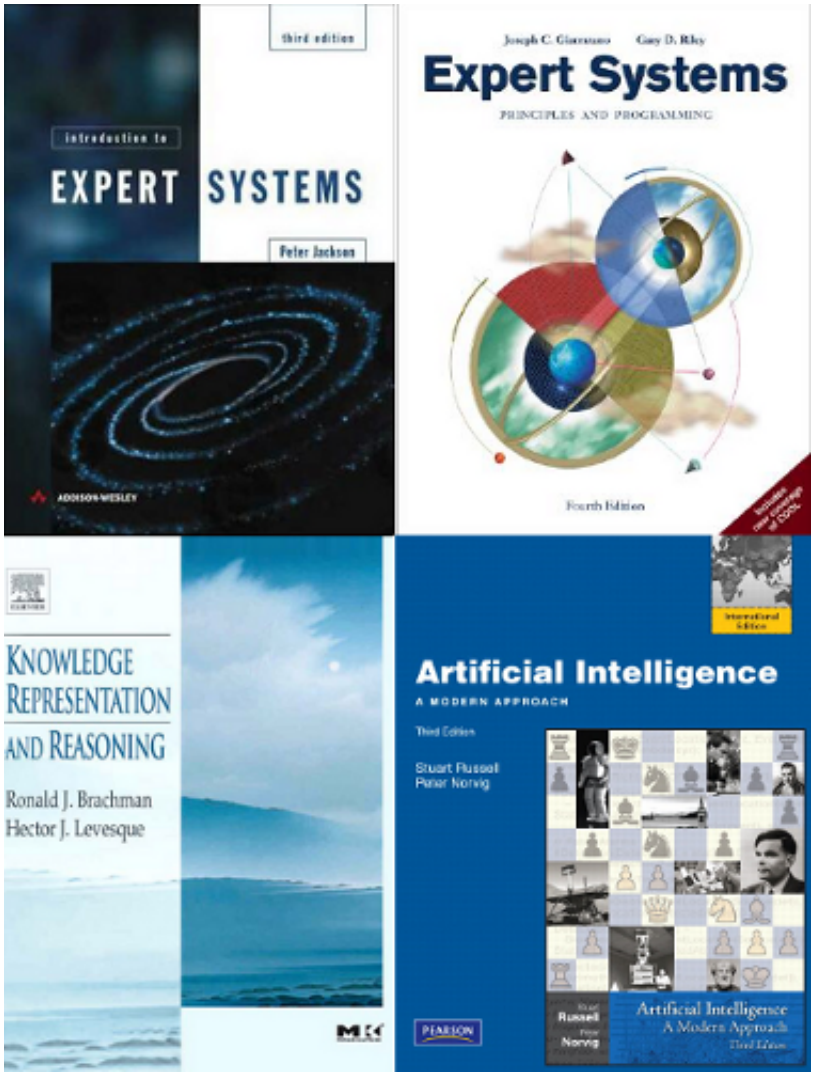
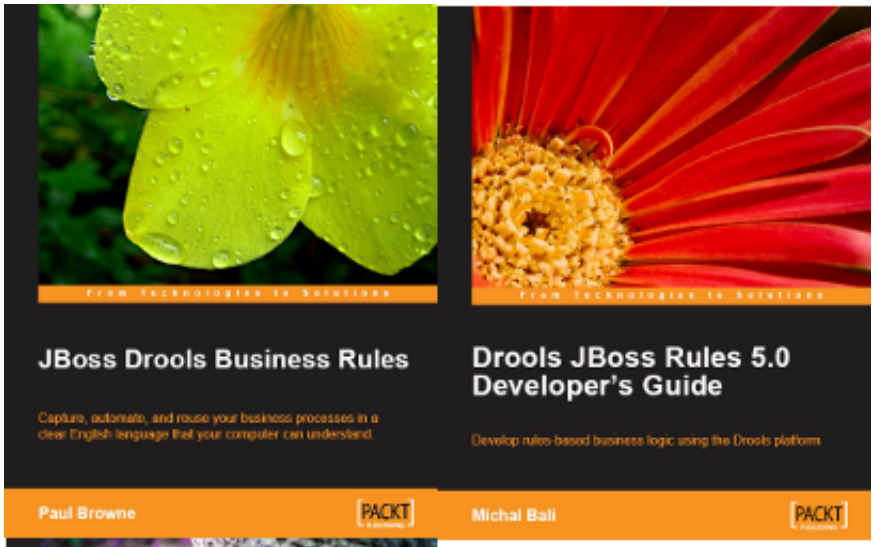


Figure 1.5. Recommended Reading

Papers Here are some recommended papers that cover some interesting areas in rule engine research.Production Matching for Large Learning Systems : Rete/UL (1993)Robert B. DoorenbosAdvances In Rete Pattern MatchingMarshall Schor, Timothy P. Daly, Ho Soo Lee, Beth R. Tibbitts (AAAI 1986)Collection-Oriented MatchAnurag Acharya and Milind Tambe (1993)The Leaps Algorithm (1990)Don BatteryGator: An Optimized Discrimination Network for Active Database Rule Condition Testing (1993)Eric Hanson , Mohammed S. Hasan**Drools Books** There are currently three Drools books, all from Packt Publishing.JBoss Drools Business RulesPaul BrownDrools JBoss Rules 5.0 Developers GuideMichali BaliDrools Developer's CookbookLucas Amador



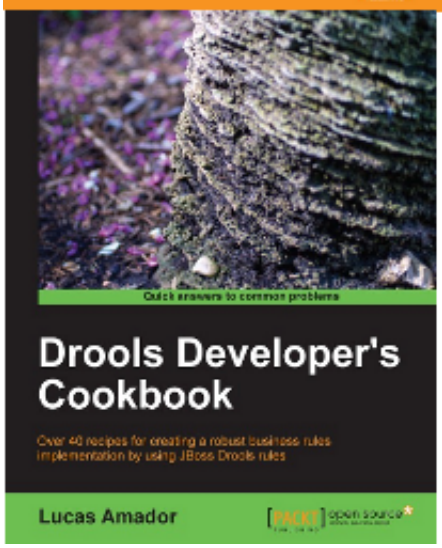


Figure 1.6. Recommended Reading

1.2. Why use a Rule Engine?

Some frequently asked questions:

1. When should you use a rule engine?
2. What advantage does a rule engine have over hand coded "if...then" approaches?
3. Why should you use a rule engine instead of a scripting framework, like BeanShell?

We will attempt to address these questions below.

1.2.1. Advantages of a Rule Engine

- Declarative Programming

  Rule engines allow you to say "What to do", not "How to do it".

The key advantage of this point is that using rules can make it easy to express solutions to difficult problems and consequently have those solutions verified. Rules are much easier to read than code.

Rule systems are capable of solving very, very hard problems, providing an explanation of how the solution was arrived at and why each "decision" along the way was made (not so easy with other of AI systems like neural networks or the human brain - I have no idea why I scratched the side of the car).

- Logic and Data Separation

  Your data is in your domain objects, the logic is in the rules. This is fundamentally breaking the OO coupling of data and logic, which can be an advantage or a disadvantage depending on your point of view. The upshot is that the logic can be much easier to maintain as there are changes in the future, as the logic is all laid out in rules. This can be especially true if the logic is cross-domain or multi-domain logic. Instead of the logic being spread across many domain objects or controllers, it can all be organized in one or more very distinct rules files.

- Speed and Scalability

  The Rete algorithm,the Leaps algorithm, and their descendants such as Drools' ReteOO, provide very efficient ways of matching rule patterns to your domain object data. These are especially efficient when you have datasets that change in small portions as the rule engine can remember past matches. These algorithms are battle proven.

- Centralization of Knowledge

  By using rules, you create a repository of knowledge (a knowledge base) which is executable. This means it's a single point of truth, for business policy, for instance. Ideally rules are so readable that they can also serve as documentation.

- Tool Integration

  Tools such as Eclipse (and in future, Web based user interfaces) provide ways to edit and manage rules and get immediate feedback, validation and content assistance. Auditing and debugging tools are also available.

- Explanation Facility

  Rule systems effectively provide an "explanation facility" by being able to log the decisions made by the rule engine along with why the decisions were made.

- Understandable Rules

  By creating object models and, optionally, Domain Specific Languages that model your problem domain you can set yourself up to write rules that are very close to natural language. They lend themselves to logic that is understandable to, possibly nontechnical, domain experts as they are expressed in their language, with all the program plumbing, the technical know-how being hidden away in the usual code.

1.2.2. When should you use a Rule Engine?

The shortest answer to this is "when there is no satisfactory traditional programming approach to solve the problem.". Given that short answer, some more explanation is required. The reason why there is no "traditional" approach is possibly one of the following:

- The problem is just too fiddle for traditional code.

  The problem may not be complex, but you can't see a non-fragile way of building a solution for it.

- The problem is beyond any obvious algorithmic solution.

  It is a complex problem to solve, there are no obvious traditional solutions, or basically the problem isn't fully understood.

- The logic changes often

  The logic itself may even be simple but the rules change quite often. In many organizations software releases are few and far between and pluggable rules can help provide the "agility" that is needed and expected in a reasonably safe way.

- Domain experts (or business analysts) are readily available, but are nontechnical.

  Domain experts often possess a wealth of knowledge about business rules and processes. They typically are nontechnical, but can be very logical. Rules can allow them to express the logic in their own terms. Of course, they still have to think critically and be capable of logical thinking. Many people in nontechnical positions do not have training in formal logic, so be careful and work with them, as by codifying business knowledge in rules, you will often expose holes in the way the business rules and processes are currently understood.

If rules are a new technology for your project teams, the overhead in getting going must be factored in. It is not a trivial technology, but this document tries to make it easier to understand.

Typically in a modern OO application you would use a rule engine to contain key parts of your business logic, ***especially the really messy parts***. This is an inversion of the OO concept of encapsulating all the logic inside your objects. This is not to say that you throw out OO practices, on the contrary in any real world application, business logic is just one part of the application. If you ever notice lots of conditional statements such as "if" and "switch", an overabundance of strategy patterns and other messy logic in your code that just doesn't feel right: that would be a place for rules. If there is some such logic and you keep coming back to fix it, either because you got it wrong, or the logic or your understanding changes: think about using rules. If you are faced with tough problems for which there are no algorithms or patterns: consider using rules.

Rules could be used embedded in your application or perhaps as a service. Often a rule engine works best as "stateful" component, being an integral part of an application. However, there have been successful cases of creating reusable rule services which are stateless.

For your organization it is important to decide about the process you will use for updating rules in systems that are in production. The options are many, but different organizations have different requirements. Frequently, rules maintenance is out of the control of the application vendors or project developers.

1.2.3. When not to use a Rule Engine

To quote a Drools mailing list regular:

	It seems to me that in the excitement of working with rules engines, that people forget that a rules engine is only one piece of a complex application or solution. Rules engines are not really intended to handle workflow or process executions nor are workflow engines or process management tools designed to do rules. Use the right tool for the job. Sure, a pair of pliers can be used as a hammering tool in a pinch, but that's not what it's designed for.	
	--Dave Hamu	

As rule engines are dynamic (dynamic in the sense that the rules can be stored and managed and updated as data), they are often looked at as a solution to the problem of deploying software. (Most IT departments seem to exist for the purpose of preventing software being rolled out.) If this is the reason you wish to use a rule engine, be aware that rule engines work best when you are able to write declarative rules. As an alternative, you can consider data-driven designs (lookup tables), or script processing engines where the scripts are managed in a database and are able to be updated on the fly.

1.2.4. Scripting or Process Engines

Hopefully the preceding sections have explained when you may want to use a rule engine.

Alternatives are script-based engines that provide the drive for "changes on the fly", and there are many such solutions.

Alternatively Process Engines (also capable of workflow) such as jBPM allow you to graphically (or programmatically) describe steps in a process. Those steps can also involve decision points which are in themselves a simple rule. Process engines and rules often can work nicely together, so they are not mutually exclusive.

One key point to note with rule engines is that some rule engines are really scripting engines. The downside of scripting engines is that you are tightly coupling your application to the scripts. If they are rules, you are effectively calling rules directly and this may cause more difficulty in future maintenance, as they tend to grow in complexity over time. The upside of scripting engines is that they can be easier to implement initially, producing results quickly, and are conceptually simpler for imperative programmers.

Many people have also implemented data-driven systems successfully in the past (where there are control tables that store meta-data that changes your applications behavior) - these can work well when the control can remain very limited. However, they can quickly grow out of control if extended too much (such that only the original creators can change the applications behavior) or they cause the application to stagnate as they are too inflexible.

1.2.5. Strong and Loose Coupling

No doubt you have heard terms like "tight coupling" and "loose coupling" in systems design. Generally people assert that "loose" or "weak" coupling is preferable in design terms, due to the added flexibility it affords. Similarly, you can have "strongly coupled" and "weakly coupled" rules. Strongly coupled in this sense means that one rule "firing" will clearly result in another rule firing, and so on; in other words, there is a clear (probably obvious) chain of logic. If your rules are all strongly coupled, the chances are that the will turn out to

be inflexible, and, more significantly, that a rule engine is an overkill. A clear chain can be hard coded, or implemented using a Decision Tree. This is not to say that strong coupling is inherently bad, but it is a point to keep in mind when considering a rule engine and the way you capture the rules. "Loosely" coupled rules should result in a system that allows rules to be changed, removed and added without requiring changes to other, unrelated rules.

1.3. Rete Algorithm

The ***Rete*** algorithm was invented by Dr. Charles Forgy and documented in his PhD thesis in 1978-79. A simplified version of the paper was published in 1982 (**http://citeseer.ist.psu.edu/context/505087/0**). The latin word "rete" means "net" or "network". The Rete algorithm can be broken into 2 parts: rule compilation and runtime execution.

The compilation algorithm describes how the Rules in the Production Memory are processed to generate an efficient discrimination network. In non-technical terms, a discrimination network is used to filter data as it propagates through the network. The nodes at the top of the network would have many matches, and as we go down the network, there would be fewer matches. At the very bottom of the network are the terminal nodes. In Dr. Forgy's 1982 paper, he described 4 basic nodes: root, 1-input, 2-input and terminal.

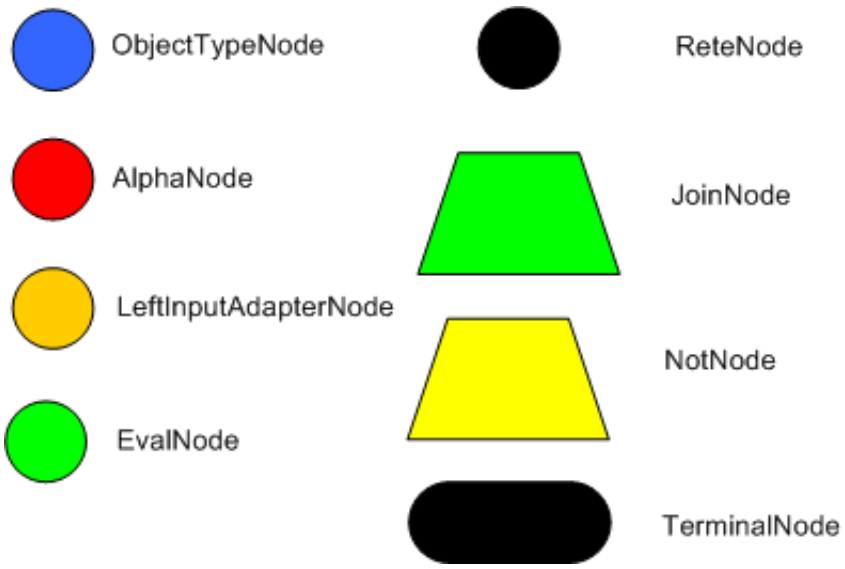


Figure 1.7. Rete Nodes

The root node is where all objects enter the network. From there, it immediately goes to the ObjectTypeNode. The purpose of the ObjectTypeNode is to make sure the engine doesn't do more work than it needs to. For example, say we have 2 objects: Account and Order. If the rule engine tried to evaluate every single node against every object, it would waste a lot of cycles. To make things efficient, the engine should only pass the object to the nodes that match the object type. The easiest way to do this is to create an ObjectTypeNode and have all 1-input and 2-input nodes descend from it. This way, if an application asserts a new Account, it won't propagate to the nodes for the Order object. In Drools when an object is asserted it retrieves a list of valid ObjectTypesNodes via a lookup in a HashMap from the object's Class; if this list doesn't exist it scans all the ObjectTypeNodes finding valid matches which it caches in the list. This enables Drools to match against any

Class type that matches with an `instanceof` check.

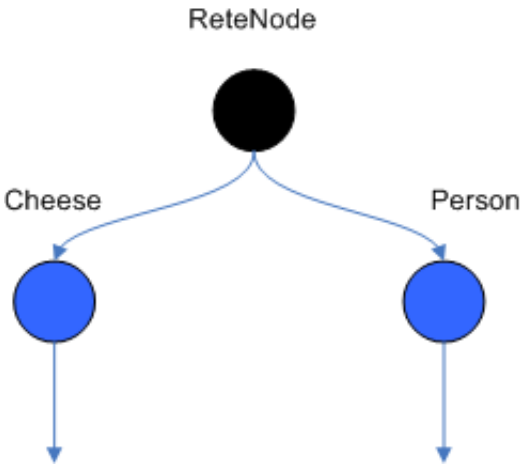


Figure 1.8. ObjectTypeNodes

ObjectTypeNodes can propagate to AlphaNodes, LeftInputAdapterNodes and BetaNodes. AlphaNodes are used to evaluate literal conditions. Although the 1982 paper only covers equality conditions, many RETE implementations support other operations. For example, `Account.name == "Mr Trout"` is a literal condition. When a rule has multiple literal conditions for a single object type, they are linked together. This means that if an application asserts an Account object, it must first satisfy the first literal condition before it can proceed to the next AlphaNode. In Dr. Forgy's paper, he refers to these as IntraElement conditions. The following diagram shows the AlphaNode combinations for Cheese(name == "cheddar", strength == "strong"):

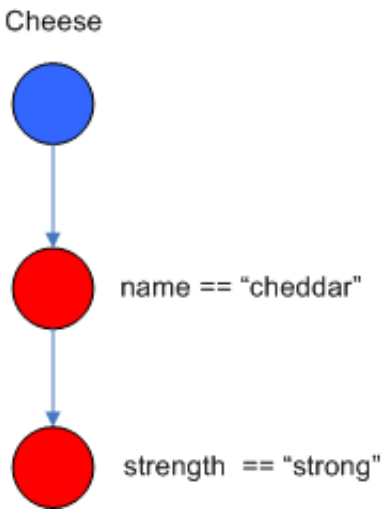


Figure 1.9. AlphaNodes

Drools extends Rete by optimizing the propagation from ObjectTypeNode to AlphaNode using hashing. Each time an AlphaNode is added to an ObjectTypeNode it adds the literal value as a key to the HashMap with the AlphaNode as the value. When a new instance enters the ObjectType node, rather than propagating to each AlphaNode, it can instead retrieve the correct AlphaNode from the HashMap,thereby avoiding unnecessary literal checks.There are two two-input nodes, JoinNode and NotNode, and both are types of BetaNodes. BetaNodes are used to compare 2 objects, and their fields, to each other. The objects may be the same or different types. By convention we refer to the two inputs as left and right. The left input for a BetaNode is generally a list of objects; in Drools this is a Tuple. The right input is a single object. Two Nodes can be used to implement 'exists' checks. BetaNodes also have memory. The left input is called the Beta Memory and

remembers all incoming tuples. The right input is called the Alpha Memory and remembers all incoming objects. Drools extends Rete by performing indexing on the BetaNodes. For instance, if we know that a BetaNode is performing a check on a String field, as each object enters we can do a hash lookup on that String value. This means when facts enter from the opposite side, instead of iterating over all the facts to find valid joins, we do a lookup returning potentially valid candidates. At any point a valid join is found the Tuple is joined with the Object; which is referred to as a partial match; and then propagated to the next node.

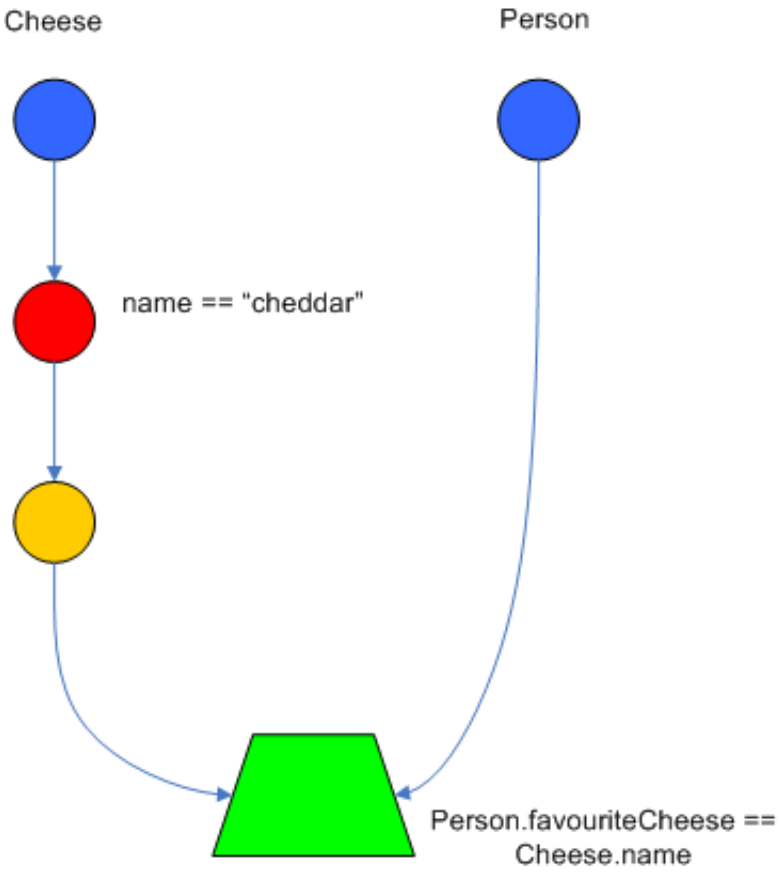


Figure 1.10. JoinNode

To enable the first Object, in the above case Cheese, to enter the network we use a LeftInputNodeAdapter - this takes an Object as an input and propagates a single Object Tuple.Terminal nodes are used to indicate a single rule having matched all its conditions; at this point we say the rule has a full match. A rule with an 'or' conditional disjunctive connective results in subrule generation for each possible logically branch; thus one rule can have multiple terminal nodes.Drools also performs node sharing. Many rules repeat the same patterns, and node sharing allows us to collapse those patterns so that they don't have to be re-evaluated for every single instance. The following two rules share the first pattern, but not the last: `rule when Cheese($cheddar : name == "cheddar") $person : Person(favouriteCheese == $cheddar) then System.out.println($person.getName() + " likes cheddar"); end``rule when Cheese($cheddar : name == "cheddar") $person : Person(favouriteCheese != $cheddar) then System.out.println($person.getName() + " does not like cheddar"); end` As you can see below, the compiled Rete network shows that the alpha node is shared, but the beta nodes are not. Each beta node has its own TerminalNode. Had the second pattern been the same it would have also been shared.

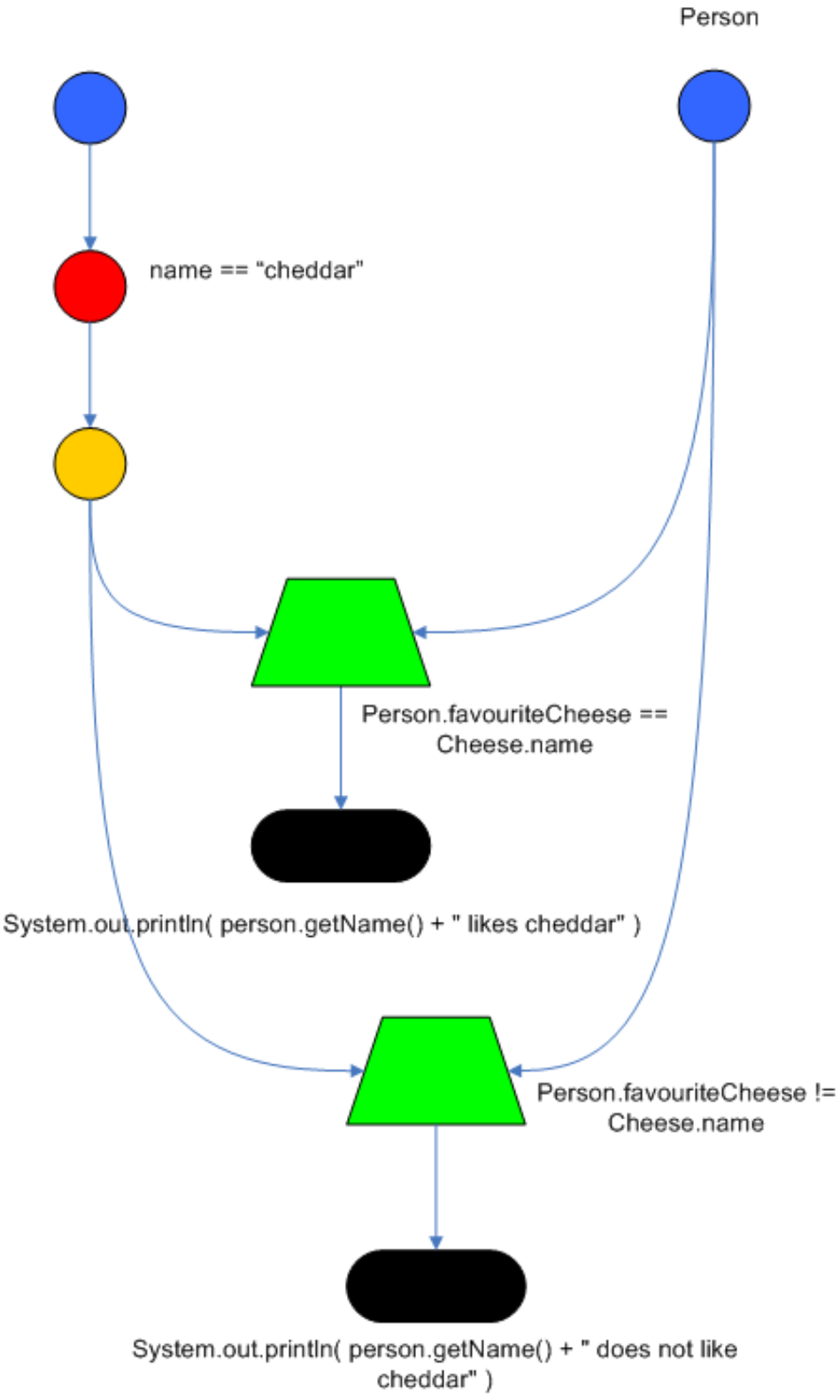


Figure 1.11. Node Sharing