ISSN:0975-9646
Narendra Kumar et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (3) , 2011, 1121-1126

Rule based programming with Drools

Narendra Kumar
BE Computer Sc., Student,
MITCOE, Pune, INDIA

Dipti D Patil
ME Computer Sc., Asst. Professor,
MITCOE, Pune, INDIA

Dr. Vijay M.Wadhai
PhD Computer Sc., Principal
MITCOE, Pune, INDIA

***Abstract*—** Imperative programming languages (such as Java) are the most widespread programming languages recently. Besides being quite easy to get familiar with them, they are also perfectly suitable for business software development. Although the productivity of imperative languages is much acclaimed, some problems are much easier to solve in a logical language. Rule based programming allows us to develop applications using declarative rules. These can simplify development in applications where such rules based knowledge is used for decision making. In this paper we will take a look at the tools techniques for developing rule based applications and discuss their strengths, capabilities, and limitations.

***Keywords*—**JBoss Rules, JSR 94, Rule Engine.

I. INTRODUCTION

RULE is a principle or regulation governing conduct, action, procedure, arrangement, etc. It is a statement that defines or constrains some aspect of the business; a business rule is intended to assert business structure or to control or influence the business's behavior. The power of business rules lies in their ability both to separate knowledge from its implementation logic and to be changed without changing source code.

```
rule "Rule Name"
when
   <Conditions>
then
   <Actions>
End
```

Many business applications have to deal with the dynamic changes of market economics. For example, applications for use in the banking and insurance industries must be able to accommodate the inevitable market changes that no one can predict or plan for during design.

As the old saying goes, "the only constant thing is change." This is certainly true for the business logic of software applications. Changes in the component(s) that implement an application's business logic can be necessary for several reasons:

- To fix code defects found during development or after deployment
- To accommodate special conditions the client initially didn't mention that the business logic should take into account
- To deal with a client's changed business objectives
- To conform to your organization's use of agile or iterative development processes

Given these possibilities, an application that can handle changes in the business logic with no major complications is highly desirable — all the more so if the developer making changes to complex if-else logic isn't the person who wrote the code.

A solution is to have a **rule engine** [1], which is basically a set of tools that enable business analysts and developers to build decision logic based on an organization's data. The rule engine evaluates and executes rules. The rule engine applies rules and actions as defined by end users without affecting how the application runs. The application is built to deal with the rules, which are designed separately.

The underlying idea of a rule engine is to externalize the business or application logic. A rule engine can be viewed as a sophisticated interpreter of *if-then* statements. The *if-then* statements are the rules. A rule is composed of two parts, a condition and an action: When the condition is met, the action is executed. The *if* portion contains conditions (such as price >=1000), and the *then* portion contains actions (such as offer discount 10%). The inputs to a rule engine are a collection of rules called a rule execution set and data objects. The outputs are determined by the inputs and may include the original input data objects with modifications, new data objects, and possible side effects.

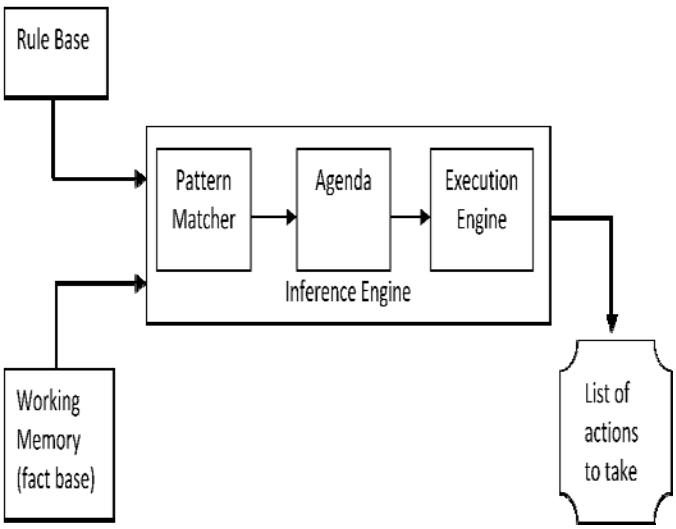

Figure 1: Rule Engine

Rule engines should be used for applications with highly dynamic business logic and for applications that allow end users to author business rules. A rule engine is a great tool for efficient decision making because it can make decisions based on thousands of facts quickly, reliably, and repeatedly.

Several rules engines are available, including commercial and open source. Commercial rules engines usually express rules in a proprietary English-like language. Others write rules using scripting languages such as Groovy or Python. Examples of rule engines include Drools, ILOG JRules, BizTalk, OpenRules, Fair Isaac Blaze Advisor, PegaRules, RulesPower, and Jess, etc. This article introduces you to the Drools engine and uses a sample program to help you understand how to use Drools as part of your business logic layer in a Java application.

1. When and where to use a rules engine?

Not all applications should use a rules engine. If your business logic code includes a bunch of if-else statements, you should consider using one. Maintaining complex Boolean logic can be a difficult task, and a rules engine can help you organize this logic. Changes are significantly less likely to introduce errors when you can express the logic using a declarative approach instead of an imperative programming language.

You should also consider a rules engine if code changes can cause major financial losses. Many organizations have strict rules about deploying compiled code in their hosting environments. For instance, if you need to modify the logic in a Java class, usually a long, tedious process must occur before the change makes it to the production environment:

- The application code must be recompiled.
- The code is dropped in a test staging environment.
- The code is inspected by data-quality auditors.
- The change is approved by the hosting environment architects.
- The change is scheduled for deployment.

Even a simple change to one line of code can cost an organization thousands of dollars. If you need to follow such strict rules and find yourself making frequent changes to your business logic code, then it would make sense to consider a rules engine. Rule engines are used in applications to replace and manage some of the business logic. They are best used in applications where the business logic is too dynamic to be managed at the source code level -- that is, where a change in a business policy needs to be immediately reflected in the application. Applications in domains such as insurance (for example, insurance rating), financial services (loans, fraud detection, claims routing and management), government (application process and tax calculations), telecom customer care and billing (promotions for long distance calls that needs to be integrated into the billing system), ecommerce (personalizing the user's experience), and so on benefit greatly from using rule engines.

Knowledge of your client can also be a factor in this decision. Even if you're working with a simple set of requirements calling for a straightforward implementation in Java code and the client has a tendency (and the financial and political resources) to add and change business logic requirements frequently during the development cycle and even after deployment. It will be better to use a rules engine from the beginning.

2. Advantages of adopting a rule-based approach:

- Rules that represent policies are easily communicated and understood.
- Rules retain a higher level of independence than conventional programming languages.
- Rules separate knowledge from its implementation logic.
- Rules can be changed without changing source code; thus, there is no need to recompile the application's code.
- Cost of production and maintenance decreases.

II. DROOLS

Drools [2],[3] is an open source rules engine, written in the Java language, that uses the **Rete algorithm** [4] to evaluate the rules. The Drools Rete implementation is called ReteOO, signifying that Drools has an enhanced and optimized implementation of the Rete algorithm for Object Oriented systems. Drools let us express our business logic rules in a declarative way. We can write rules using a non-XML native language that is quite easy to learn and understand. And we can embed Java code directly in a rules file. Drools also have other advantages. It is:

- Supported by an active community
- Easy to use
- Quick to execute
- Gaining popularity among Java developers
- Compliant with the Java Rule Engine API (JSR 94)
- Free

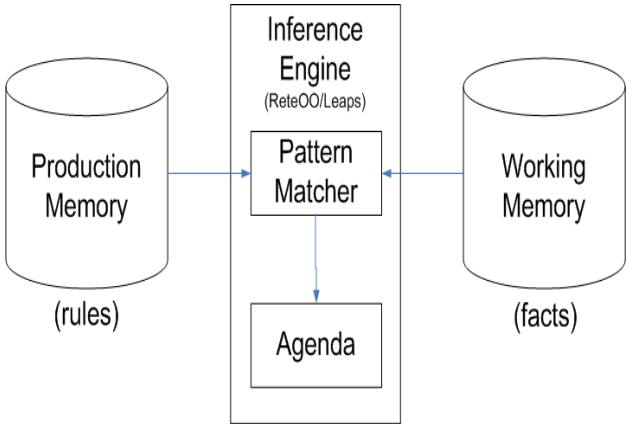

Figure 2: A Basic Rete network

Drools is a Rule Engine that uses the Rule Based approached to implement an Expert System and is more correctly classified as a Production Rule System. A Production Rule System is

Turing complete with a focus on knowledge representation to express propositional and first order logic in a concise, non ambiguous and declarative manner. The brain of a Production Rules System is an Inference Engine (see Fig 2) that is able to scale to a large number of rules and facts. The Inference Engine matches facts and data, against Production Rules, also called Productions or just Rules, to infer conclusions which result in actions. A Production Rule is a two-part structure using First Order Logic for knowledge representation.

when
<Conditions>
then
<Actions>

The Rules are stored in the Production Memory and the facts that the Inference Engine matches against 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.

A Production Rule System's Inference Engine is statefull and able to enforce truthfulness - called Truth Maintenance. A logical relationship can be declared by actions which mean the action's state depends on the inference remaining true; when it is no longer true the logical dependent action is undone. The "Honest Politician" is an example of Truth Maintenance, which always ensures that hope can only exist for a democracy while we have honest politicians.

```
when
   an honest Politician exists
then
   logically assert Hope

when
  Hope exists
then
  print "Hurrah!!! Democracy Lives"

when
  Hope does not exist
then
  print "Democracy is Doomed"
```

There are two methods of execution for a Production Rule Systems - Forward Chaining and Backward Chaining; systems that implement both are called Hybrid Production Rule Systems. Understanding these two modes of operation are key to understanding why a Production Rule System is different and how to get the best from them. Forward chaining is 'data-driven' and thus reactionary - facts are asserted into the working memory which results in one or more rules being concurrently true and scheduled for execution by the Agenda - we start with a fact, it propagates and we end in a conclusion. Drools is a forward chaining engine.

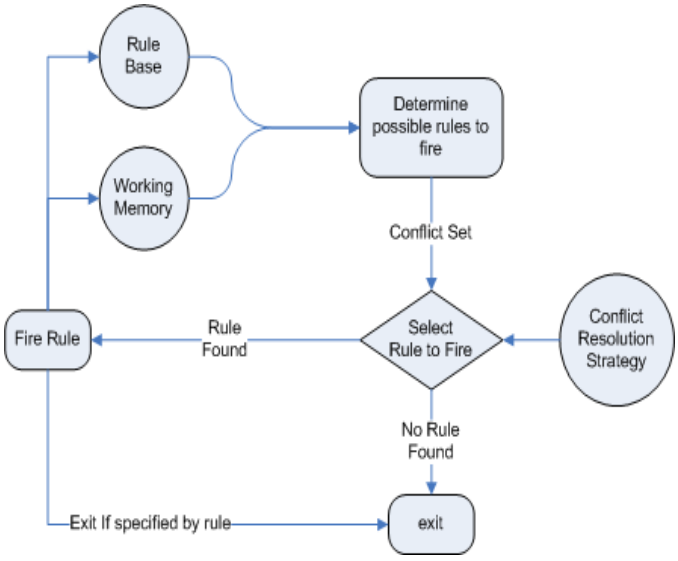

Figure 3: Forward chaining

III. RETE ALGORITHM

The word RETE is Latin for "net" meaning network. The RETE algorithm can be broken into two parts: rule compilation and runtime execution.

The compilation algorithm describes how the Rules in the Production Memory to generate an efficient discrimination network. In non-technical terms, a discrimination network is used to filter data. The idea is to filter data as it propagates through the network. At the top of the network the nodes 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. There are four basic nodes: root, 1-input, 2-input and terminal.

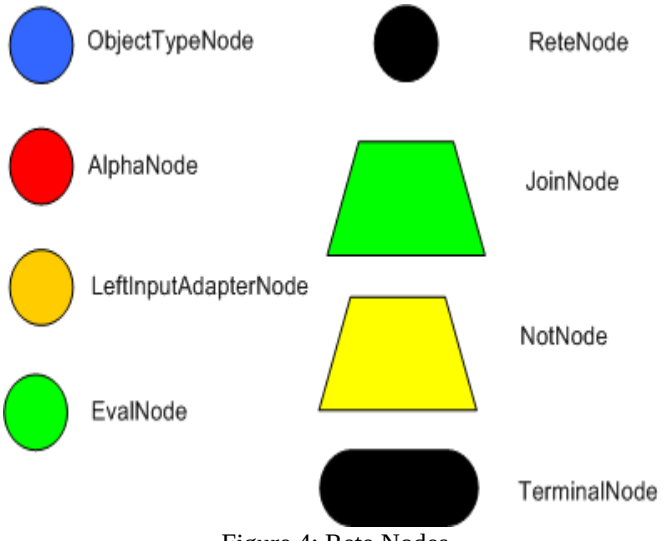

Figure 4: Rete Nodes

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 descended 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 ObjectTypde nodes finding valid matches which it caches in the list. This enables Drools to match against any Class type that matches with an instance of check.

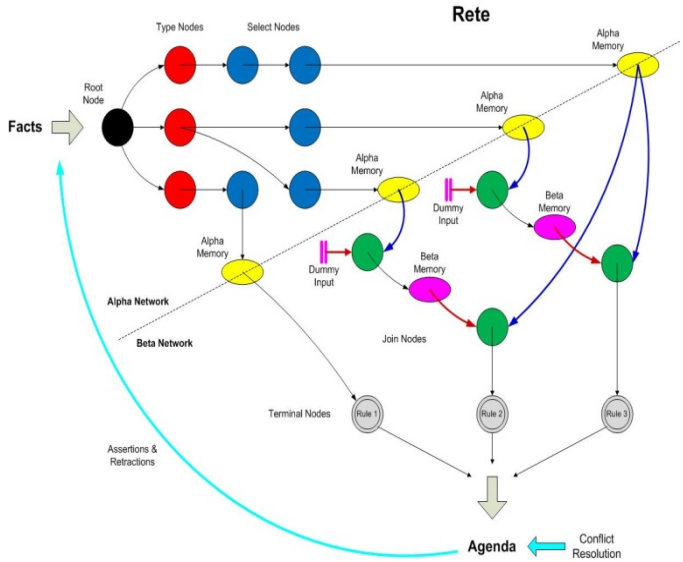

Figure 5: Rete topography

IV. An Example: Getting a Car Loan

This article shows how to use Drools as part of the business logic layer in a sample Java application. To follow along, you should be familiar with developing and debugging Java code using the Eclipse IDE [7]. And you should be familiar with the JUnit testing framework and know how to use it within Eclipse.

The process of determining how and when to give a car loan is complex and can change quite often. We need to consider an applicant's credit score, income, and down payment, among other things. Therefore it is good candidate for use with Drools. First we'll define the CarBuyer class.

```
package com.carloan;

public class CarBuyer {
      int  creditScore;
      int  downPayment;
      String  name;

      public CarBuyer(String buyerName, int creditScore, int downPayment) {
            name = buyerName;
            creditScore = creditScore;
            downPayment = downPayment;
      }

      public String getName() {
            return  name;
      }

      public int getCreditScore(){
            return  creditScore;
      }

      public int getDownPayment(){
            return  downPayment;
      }
}
```

Next, we'll need a class that sets up and runs the rules. Here is our LoanDeterminizer:

```
package com.carloan;
import org.drools.jsr94.rules.RuleServiceProviderImpl;

import javax.rules.RuleServiceProviderManager;
import javax.rules.RuleServiceProvider;
import javax.rules.StatelessRuleSession;
import javax.rules.RuleRuntime;
import javax.rules.admin.RuleAdministrator;
import javax.rules.admin.LocalRuleExecutionSetProvider;
import javax.rules.admin.RuleExecutionSet;
import java.io.InputStream;
import java.util.ArrayList;

public class LoanDeterminizer {
      private boolean  okToGiveLoan;
      private CarBuyer  carBuyer;
      private int  costOfCar;
      public boolean giveLoan(CarBuyer h, int costOfCar) {
            okToGiveLoan = true;
            carBuyer = h;
            costOfCar = costOfCar;

            ArrayList<Object> objectList = new ArrayList<Object>();
            objectList.add(h);
            objectList.add(costOfCar);
            objectList.add(this);

            return _okToGiveLoan;
      }

      public CarBuyer getCarBuyer() { return _carBuyer; }
      public int getCostOfCar() { return  costOfCar; }
      public boolean getOkToGiveLoan() { return  okToGiveLoan; }
      public double getPercentDown() {
            return(double)(carBuyer.getDownPayment()/costOfCar);
          }

              private final String RULE_URI = "LoanRules.drl";
 // this is the file name our Rules are contained in
      public LoanDeterminizer() throws Exception     {
      prepare();
  }
```

```
    private final String  RULE_SERVICE_PROVIDER = "http://dro
ols.org/";

   private StatelessRuleSession  statelessRuleSession;
   private RuleAdministrator  ruleAdministrator;

   private boolean  clean = false;

   protected void finalize() throws Throwable
   {
      if (! clean)  { cleanUp(); }
   }

   private void prepare() throws Exception
   {
      RuleServiceProviderManager.registerRuleServiceProvider(
            RULE_SERVICE_PROVIDER, RuleServiceProviderImpl.
class );

      RuleServiceProvider ruleServiceProvider  =
            RuleServiceProviderManager.getRuleServiceProvider(
                                    RULE_SERVICE_PROVIDER);

     ruleAdministrator = ruleServiceProvider.getRuleAdministrator( );

      LocalRuleExecutionSetProvider ruleSetProvider =
           ruleAdministrator.getLocalRuleExecutionSetProvider(null);

      InputStream rules  =
              Exchange.class.getResourceAsStream(RULE_URI);

   RuleExecutionSet ruleExecutionSet =
            ruleSetProvider.createRuleExecutionSet(rules, null);

   ruleAdministrator.registerRuleExecutionSet(RULE_URI,
                        ruleExecutionSet, null);


   RuleRuntime ruleRuntime  =
                    ruleServiceProvider.getRuleRuntime();

      statelessRuleSession =
(StatelessRuleSession) ruleRuntime.createRuleSession(RULE_URI,
                 null, RuleRuntime.STATELESS_SESSION_TYPE );
   }

   public void cleanUp() throws Exception
   {
      clean = true;
      statelessRuleSession.release( );
      ruleAdministrator.deregisterRuleExecutionSet(RULE_URI,
       null);
   }

}
```

The testRules class:

```
package com.carloan;

import junit.framework.TestCase;

public class TestRules extends TestCase {

   public void test_poor_credit_rating_gets_no_loan()
                throws Exception {
      LoanDeterminizer ld = new LoanDeterminizer();
      CarBuyer h = new CarBuyer("buyerName", 100, 20000);

      boolean result = ld.giveLoan(h, 150000);
      assertFalse(result);

      ld.cleanUp();
   }

}
```

Now we can finally write our rules in LoanRules.drl:

```
package com.carloan

rule "High credit score always gets a loan"
    salience 1
    when
       buyer : CarBuyer(creditScore >= 700)
       loan_determinizer : LoanDeterminizer(carBuyer == buyer)
    then
       System.out.println(buyer.getName() + " has a credit rating to get
                                the loan no matter the down payment.");
       loan_determinizer.setOkToGiveLoan(true);
end

rule "Middle credit score fails to get a loan with small down payment"
    salience 0
    when
       buyer : CarBuyer(creditScore >= 400 && creditScore < 700)
       loan_determinizer : LoanDeterminizer(carBuyer == buyer &&
                percentDown < 0.20)
    then
       System.out.println(buyer.getName() + " has a credit rating to get
                                the loan but not enough down payment.");
       loan_determinizer.setOkToGiveLoan(false);
end

rule "Poor credit score never gets a loan"
 salience 2
 when
  buyer : CarBuyer(creditScore < 400)
  loan_determinizer : LoanDeterminizer(carBuyer == buyer)
 then
  System.out.println(buyer.getName() + " has too low a credit rating
                            to get the loan.");
  loan_determinizer.setOkToGiveLoan(false);

end
```

The string following "rule" is the rule's name. Salience is one of the ways Drools performs conflict resolution. Finally, the first two lines tell it that buyer is a variable of type CarBuyer with a credit score of less than 400 and loan_determinizer is the LoanDeterminizer passed in with the object list where the carBuyer is what we've called buyer in our rule. If either of those conditions fails to match, this rule is skipped. An important aspect of a rule is the optional salience attribute. It is used to let the rules execution engine know the order in which it should fire the consequence statements of your rules. The consequence statements of the rule with the highest salience value are executed first; the consequence statements of the rule with the second-highest salience value are executed second, and so on. This is important when we need our rules to be fired in a predefined order.

V. CONCLUSION

Using a rules engine can significantly reduce the complexity of components that implement the business-rules logic in our Java applications. An application that uses a rules engine to express rules using a declarative approach has a higher chance of being more maintainable and extensible than one that doesn't. Drools is a powerful and flexible rules engine implementation. Using Drools' features and capabilities, we should be able to implement the complex business logic of our application in a declarative manner. Drools make learning and using declarative programming quite easy for Java developers.

The Drools classes that this article showed are Drools-specific. If we were to use another rules engine implementation with the sample program, the code would need a few changes. Because Drools is JSR 94-compliant, we could use the Java Rule Engine API (as specified in JSR 94) to interface with Drools-specific classes. If we use this API, then we can change your rules engine implementation to a different one without needing to change the Java code, as long as this other implementation is also JSR 94-compliant. JSR 94 does not address the structure of the rules file that contains your business rules. The file's structure would still depend on the rules engine implementation.

REFERENCES

[1] http://en.wikipedia.org/wiki/Business_rules_engine
[2] Paul Browne, "JBoss Drools Business Rules," **ISBN :** 1847196063 **ISBN 13 :** 978-1-847196-06-4, PACKT publishing.
[3] Michal Bali, "Dools JBoss Rules 5.0 Developer's Guide" **ISBN :** 1847195644, **ISBN 13 :** 978-1-847195-64-7, PACKT publishing.
[4] http://downloads.jboss.com/drools/docs/5.1.1.34858.FINAL/drools-introduction/html/index.html.
[5] http://blog.athico.com/2010/03/fosdem-50-minute-introduction-into.html
[6] http://en.wikipedia.org/wiki/Drools.
[7] http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/helios/SR2/eclipse-jee-helios-SR2-win32.zip
[8] http://www.cis.temple.edu/~ingargio/cis587/readings/rete.html
[9] http://downloads.jboss.com/drools/docs/5.1.1.34858.FINAL/drools-integration/html_single/index.html
[10] http://downloads.jboss.com/drools/docs/5.1.1.34858.FINAL/drools-flow/html_single/index.html
[11] http://www.mastertheboss.com/jbpm/45-jboss-drools-1.html
[12] http://www.jessrules.com/jess/docs/52/rete.html
[13] http://www.jbug.jp/trans/jboss-rules3.0.2/ja/html/ch01s06.html
[14] http://en.wikipedia.org/wiki/Rete_algorithm
[15] http://en.wikipedia.org/wiki/Business_rule_management_system

AUTHOR PROFILES

Narendra kumar is a student in Department of Computer Engineering in MIT College of Engineering, Pune, India. He is pursuing his Bachelor's degree in Computer Science and will be completing his engineering in June, 2011. . Email : narendra210389@gmail.com

Prof. Dipti D Patil is pursuing her PhD in computer sc from university of Pune. She has received M.E. degree in Computer Engineering from Mumbai University, India in 2008 and B.E. degree in Computer Engineering from Mumbai University in 2002. She has worked as Head & Assistant Professor in Computer Engineering Department in Vidyavardhini's College of Engineering & Technology, Vasai. She is currently working as Assistant Professor in MITCOE, Pune. Her Research interests include Data mining and Body Area Network.

Dr. Vijay M.Wadhai received his Ph.D. degree from Amravati University in 2007, M.E. from Gulbarga University in 1995 and B.E. from Nagpur University in 1986. He has experience of 25 years which includes both academic (17 years) and research (8 years). He has been working as a Principal of MITCOE, Pune and simultaneously handling the post of Director - Research and Development, Intelligent Radio Frequency (IRF) Group, Pune (from 2009). He is currently guiding 12 students for their PhD work in both Computers and Electronics & Telecommunication area. His research interest includes Data Mining, Natural Language processing, Cognitive Radio and Wireless Network, Spectrum Management, Wireless Sensor Network, VANET, Body Area Network, ASIC Design - VLSI. He is a member of ISTE, IETE, IEEE, IES and GISFI (Member Convergence Group), India.