RULE SOLVER™

Constraint Programming

with

OpenRules® 6.3.1

USER MANUAL

OpenRules, Inc.

www.openrules.com

May-2014
# Table of Contents

**Introduction** .................................................................................. 4

**Document Conventions** .................................................................. 5

**Rule Solver as Inferential Rule Engine** ........................................... 6

**Simple Examples** ........................................................................... 6
  Simple Decision Model “DecisionHello” ........................................... 6
  Decision Model ............................................................................. 7
  Solution with Rule Engine ............................................................ 8
  Solution with Rule Solver .............................................................. 11
  Simple Decision Model “DecisionLoan” ......................................... 14
  Decision Model ............................................................................ 14
  Solution with Rule Engine ............................................................ 19
  Solution with Rule Solver .............................................................. 20

**Decision Modeling with Rules Solver** ........................................... 22
  Rule Solver™ Benefits .................................................................... 22
  How Rule Solver™ Works .............................................................. 22
  Generating CSP ............................................................................ 23
  Adding Constrained Variables ...................................................... 24
  Adding Constraints ....................................................................... 24
  Posting Data Constraints .............................................................. 25
  Posting Problem Constraints ......................................................... 25
  Solving the Problem ..................................................................... 26
  Using Templates ............................................................................ 27
  Using JSR-331 .............................................................................. 28
  Decision Execution Reports ........................................................... 28

**OpenRules Solver Execution Report** ............................................. 29
  Decision "DetermineLoanPreQualificationResults" .......................... 29
  Wed Apr 23 16:41:58 EDT 2014 ....................................................... 29
  === Assigning Data To Decision Variables === ............................... 29
  === Posting If-Then Constraints === ............................................. 30
  Posted If-Then Constraints .......................................................... 30
  Decision Output ............................................................................ 33
  Implementation Restrictions and Future Improvements ................ 33

**Rule Solver as a Business-Oriented Constraint Solver** .................. 34

**Constraint Satisfaction Problem (CSP)** ........................................ 34
  Formal Definition .......................................................................... 34
  Major CP Concepts ....................................................................... 35
  Introductory Example “SEND+MORE=MONEY” ............................. 36
  Excel-based Decision Model ......................................................... 36
  Pure Java Solution ........................................................................ 40
  Solving Arithmetic Problems ....................................................... 41
INTRODUCTION

Today **Constraint Programming (CP)** has become a leading technique for solving complex constraint satisfaction and optimization problems in manufacturing, telecom, logistics, finance, and other industries. Among such problems are job scheduling, resource allocation, planning, product configuration, and other decision support problems with many business constraints. CP provides a great foundation for the development of smart optimization and decision support engines. There are multiple powerful commercial and open source **constraint solvers** available on the market today.

OpenRules® Business Decision Management System (BDMS) includes a special component called a **Rule Solver™** that empowers OpenRules® with constraint programming functionality. Rule Solver™ can be used for two major purposes:

1) **Rule Solver™ as an inferential rule engine for decision models**
   - An alternative to the standard OpenRules® sequential rule engine that executes decision models in a way similar to famous RETE-based rule engines (no needs for rule ordering)
   - A powerful mechanism for consistency validation of OpenRules® decision models.

2) **Rule Solver™ as a business-oriented constraint solver**
   - An ability to represent constraint satisfaction problems using Excel-based decision tables oriented to business users
   - An ability to solve constraint satisfaction problems with any JSR-331\(^1\) compliant constraint solver.

This user manual explains how to install and use Rule Solver™. It is aimed at developers of real-world decision models that need a more sophisticated mechanism to compare with traditional rule engines. Rule Solver™ includes a

---

\(^1\) JSR-331 “Constraint Programming API” is a Java Community Process standard that was awarded the Most Innovative JSR Award at Java One, 2010.
variety of templates that allow business analysts (not necessarily familiar with CP or even Java) to define their own scheduling, resource allocation, configuration, and other constraint satisfaction problems and use standard CP solving methods to find their solutions.

**DOCUMENT CONVENTIONS**

The regular Century Schoolbook font is used for information that is prescriptive by this specification.

*The italic Century Schoolbook font is used for notes clarifying the text*

The Courier New font is used for code examples.
Rule Solver™ can be used as an inferential Rule Engine that can execute OpenRules® decision models. It provides an alternative to the standard OpenRules® sequential rule engine. At the same time it provides a powerful validation mechanism that automatically checks OpenRules® decision models for consistency and completeness.

The decision models defined using traditional OpenRules® decision tables can be executed in two modes:

1. “Execute” mode that uses a regular OpenRules® rule engine. In this, default, mode a user is expected to explicitly specify the order of rules within a decision table and the execution order of decision tables inside decisions.

2. “Solve” mode that uses Rule Solver™. In this mode the order of rules within a decision table and between decision tables does not matter and can be resolved automatically. Functionally this mode is similar to famous RETE-based rule engines with no needs for rules ordering.

Thus, the same decision model expressed in business terms can serve as an input for both rule engines: a regular (“sequential”) rule engine and an inferential (constraint-based) rule engine.

Simple Examples

The following examples demonstrate how to apply sequential Rule Engine and inferential Rule Solver to the same decision models.

Simple Decision Model “DecisionHello”

In this example we will develop a simple application that should decide how to greet a customer during different times of the day. The proper decision model might be a part of an interactive voice response (IVR) system. For example, if a
customer Robinson is a married woman and local time is 14:25, we want our decision to produce a greeting like "Good Afternoon, Mrs. Robinson!". To make this example a little bit more complicated we will force the application to greet children with a greeting like "Good Afternoon, Little Robinson!".

**Decision Model**

We will use Excel to represent decisions, related decision tables, and several test cases. We will start with the main table of the type “Decision” that consists of 3 sub-decisions:

<table>
<thead>
<tr>
<th>Decision DetermineCustomerGreeting</th>
<th>Execute Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define Age Group</td>
<td>DefineAgeGroup</td>
</tr>
<tr>
<td>Define Greeting Word</td>
<td>DefineGreeting</td>
</tr>
<tr>
<td>Define Salutation Word</td>
<td>DefineSalutation</td>
</tr>
</tbody>
</table>

The first sub-decision “Define Age Group” will be implemented using the following decision table:

<table>
<thead>
<tr>
<th>DecisionTable DefineAgeGroup</th>
<th>Condition</th>
<th>Condition</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>Age</td>
<td>Age Group</td>
<td></td>
</tr>
<tr>
<td>&gt;= 0</td>
<td>&lt;= 5</td>
<td>Little</td>
<td></td>
</tr>
<tr>
<td>&gt; 5</td>
<td>&lt;= 20</td>
<td>Young</td>
<td></td>
</tr>
<tr>
<td>&gt; 20</td>
<td></td>
<td>Adult</td>
<td></td>
</tr>
</tbody>
</table>

The second sub-decision “Define Greeting Word” will be implemented using the following decision table:

<table>
<thead>
<tr>
<th>DecisionTable DefineGreeting</th>
<th>Condition</th>
<th>Condition</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Hour</td>
<td>Current Hour</td>
<td>Greeting</td>
<td></td>
</tr>
<tr>
<td>&gt;= 0</td>
<td>&lt;= 11</td>
<td>Is Good Morning</td>
<td></td>
</tr>
<tr>
<td>&gt; 11</td>
<td>&lt;= 17</td>
<td>Is Good Afternoon</td>
<td></td>
</tr>
<tr>
<td>&gt; 17</td>
<td>&lt;= 22</td>
<td>Is Good Evening</td>
<td></td>
</tr>
<tr>
<td>&gt; 22</td>
<td>&lt;= 24</td>
<td>Is Good Night</td>
<td></td>
</tr>
</tbody>
</table>
The third sub-decision “Define Salutation Word” will be implemented using the following decision table:

<table>
<thead>
<tr>
<th>Gender</th>
<th>Marital Status</th>
<th>Age Group</th>
<th>Salutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is</td>
<td>Female</td>
<td>Is Not</td>
<td>Little</td>
</tr>
<tr>
<td>Is</td>
<td>Female</td>
<td>Is Single</td>
<td>Little</td>
</tr>
<tr>
<td>Is</td>
<td>Male</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The proper Glossary for this model can be defined as follows:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Business Concept</th>
<th>Attribute</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gender</td>
<td>Customer</td>
<td>gender</td>
<td>Male,Female</td>
</tr>
<tr>
<td>Marital Status</td>
<td></td>
<td>maritalStatus</td>
<td>Single,Married</td>
</tr>
<tr>
<td>Age</td>
<td></td>
<td>age</td>
<td>0-120</td>
</tr>
<tr>
<td>Age Group</td>
<td></td>
<td>ageGroup</td>
<td>Little,Young,Adult</td>
</tr>
<tr>
<td>Current Hour</td>
<td></td>
<td>hour</td>
<td>0-24</td>
</tr>
<tr>
<td>Greeting</td>
<td>Response</td>
<td>greeting</td>
<td>Good Morning,Good Afternoon,Good Evening, Good Night</td>
</tr>
<tr>
<td>Salutation</td>
<td></td>
<td>salutation</td>
<td>Mr.,Ms.,Mrs.,Little</td>
</tr>
</tbody>
</table>

We will assume that the data for our model comes from a Java as defined by the following table of the type “DecisionObject”:

<table>
<thead>
<tr>
<th>DecisionObject</th>
<th>decisionObjects</th>
<th>Business Concept</th>
<th>Business Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>:= decision.get(&quot;customer&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Response</td>
<td>:= decision.get(&quot;response&quot;)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This completes the definition of our decision model. Now we find solutions for this model starting with a regular (sequential) OpenRules® Rule Engine.

**Solution with Rule Engine**

The standard OpenRules® installation comes with the proper decision project “DecisionHelloCP” in the workspace “openrules.solver”. This project has a Java
package “hello” with two Java classes “Customer” and “Response” that are simple Java beans with the following organization:

```java
public class Customer {
    String name;
    String maritalStatus;
    String gender;
    int age;
    String ageGroup;
    int hour;
    // getters and setters
}

public class Response {
    String greeting;
    String salutation;
    String result;
    // getters and setters
}
```

The main Java class Main.java contains one method “main” that creates test-instances of the classes Customer and Response, puts them to the instance of Decision, and executes this decision:

```java
public class Main {

    public static void main(String[] args) {
        String fileName = "file:rules/main/Decision.xls";
        System.setProperty("OPENRULES_MODE", "Execute");
        Decision decision =
            new Decision("DetermineCustomerGreeting",fileName);

        Customer customer = new Customer();
        customer.setName("Robinson");
        customer.setGender("Female");
        customer.setMaritalStatus("Married");
        customer.setAge(4);
        customer.setHour(16);
        Response response = new Response();
        decision.put("customer", customer);
        decision.put("response", response);
        decision.put("trace","On");
        decision.execute();
        out.println("Decision: "+ response.getGreeting()
            + ", " + response.getSalutation()
            + " " + customer.getName() + "!");
    }
```
This code uses a predefined OpenRules® class “Decision” that extends HashMap and allows a user to put and get any object to the decision using keywords like “customer”.

The above statement

```java
System.setProperty("OPENRULES_MODE", "Execute");
```

reinforces the fact that we will use the default execution mode that is based on the standard OpenRules® rule engine. After execution of this code we will receive the following results:

```
*** Decision DetermineCustomerGreeting ***
Decision has been initialized
Decision DetermineCustomerGreeting: Define Greeting Word
Conclusion: Greeting Is Good Afternoon
Decision DetermineCustomerGreeting: Define Age Group
Conclusion: Age Group Is Little
Decision DetermineCustomerGreeting: Define Salutation Word
Conclusion: Salutation Is Little
Decision has been finalized

Decision: Good Afternoon, Little Robinson!
```

There are many other examples and more powerful decision table types described in the OpenRules® User Manual. However, the standard OpenRules® is rule engine sequential and relies on the strictly defined execution order of rules inside the table “Decision”. Let’s change this order as follows:

<table>
<thead>
<tr>
<th>Decision DetermineCustomerGreeting</th>
<th>Decide Greeting Word</th>
<th>DefineSalutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision DetermineCustomerGreeting</td>
<td>Define Greeting Word</td>
<td>DefineSalutation</td>
</tr>
<tr>
<td>Define Greeting Word</td>
<td>DefineGreeting</td>
<td></td>
</tr>
<tr>
<td>Define Salutation Word</td>
<td>DefineSalutation</td>
<td></td>
</tr>
<tr>
<td>Define Age Group</td>
<td>DefineAgeGroup</td>
<td></td>
</tr>
</tbody>
</table>

As you can see, now the standard rule engine will execute the rules “DefineAgeGroup” after(!) the rules “DefineSalutation” which use the decision variable “Age Group” defined by “DefineAgeGroup”. So, the variable will remain undefined (“null”) and the execution results will be quite bad:
 *** Decision DetermineCustomerGreeting ***
Decision has been initialized
Decision DetermineCustomerGreeting: Define Greeting Word
Conclusion: Greeting Is Good Afternoon
Decision DetermineCustomerGreeting: Define Salutation Word
Decision DetermineCustomerGreeting: Define Age Group
Conclusion: Age Group Is Little
Decision has been finalized
Decision: Good Afternoon, null Robinson!

Now we find a solution for the same decision model using OpenRules® Rule Solver that resolves rules sequencing issues automatically.

Solution with Rule Solver

The project “DecisionHelloCP” include another Java class “MainCP” that uses the same classes Customer and Response but with the following main-method:

```java
public class Main {

    public static void main(String[] args) {
        String fileName = "file:rules/main/Decision.xls";
        System.setProperty("OPENRULES_MODE", "Solve"); // !!!

        Decision decision =
                new Decision("DetermineCustomerGreeting",fileName);

        Customer customer = new Customer();
        customer.setName("Robinson");
        customer.setGender("Female");
        customer.setMaritalStatus("Married");
        customer.setAge(4);
        customer.setHour(16);
        Response response = new Response();
        decision.put("customer", customer);
        decision.put("response", response);

        decision.put("trace", "On");
        decision.execute();

        out.println("Decision: "+ response.getGreeting()
                + ", " + response.getSalutation()
                + " " + customer.getName() + "{quote}");
    }
}
```

This code is exactly the same as above with only difference in this statement:

```java
System.setProperty("OPENRULES_MODE", "Solve");
```
It forces OpenRules® to use Rule Solver™ instead of the default rule engine. In this case the execution results of the same decision model (with the latest order of sub-decisions) will look as follows:

```plaintext
*** Decision DetermineCustomerGreeting ***
Create RuleSolver
JSR-331 Implementation based on Constrainer 5.4 (light)

=== Rule Solver (version 6.2.0) ===
addConstrainedVariables
Decision has been initialized with RuleSolver

=== Initial Problem Variables: ===
Marital Status[Single,Married]
Greeting[Good Morning,Good Afternoon,Good Evening,Good Night]
Age[0..120]
Age Group[Little,Young,Adult]
Gender[Male,Female]
Salutation[Mr.,Ms.,Mrs.,Little]
Current Hour[0..24]

Decision DetermineCustomerGreeting: Define Greeting Word
Decision DetermineCustomerGreeting: Define Salutation Word
Decision DetermineCustomerGreeting: Define Age Group

=== After Assigning Data: ===
Marital Status[Married]
Greeting[Good Morning,Good Afternoon,Good Evening,Good Night]
Age[4]
Age Group[Little,Young,Adult]
Gender[Female]
Salutation[Mr.,Ms.,Mrs.,Little]
Current Hour[16]

=== After Posting Constraints: ===
Marital Status[Married]
Greeting[Good Afternoon]
Age[4]
Age Group[Little]
Gender[Female]
Salutation[Little]
Current Hour[16]

=== Solve ===
Decision: Good Afternoon, Little Robinson!
```

As you can see, in spite of the “wrong” order of rules, Rule Solver™ managed to produce the correct solution “Good Afternoon, Little Robinson!” instead of the previous “Good Afternoon, null Robinson!”.

The produced execution trace explains how Rule Solver™ actually works:
Step 1. First Rule Solver™ creates an instance of the predefined class “RuleSolver” that is based on “JSR-331 Implementation based on Constrainer 5.4 (light)”

Step 2. Then Rule Solver™ generates a constraint satisfaction problem by adding constrained variables (based on the Glossary and actual business objects) and constraints (based on decision tables). So, it “adds Constrained Variables” and shows their initial state as:

- Marital Status: [Single, Married]
- Greeting: [Good Morning, Good Afternoon, Good Evening, Good Night]
- Age: [0..120]
- Age Group: [Little, Young, Adult]
- Gender: [Male, Female]
- Salutation: [Mr., Ms., Mrs., Little]
- Current Hour: [0..24]

Step 3. When Rule Solver™ executes the sub-decisions from the table “Decision”

- Decision DetermineCustomerGreeting: Define Greeting Word
- Decision DetermineCustomerGreeting: Define Salutation Word
- Decision DetermineCustomerGreeting: Define Age Group

it actually creates the proper constraints but does not post (activate) them yet.

Step 4. Then Rule Solver™ assigns data to decision variables that are already known. In our case, those variables are “Marital Status”, “Age”, “Gender”, and “Current Hour”.

=== After Assigning Data:
- Marital Status: [Married]
- Greeting: [Good Morning, Good Afternoon, Good Evening, Good Night]
- Age: [4]
- Age Group: [Little, Young, Adult]
- Gender: [Female]
- Salutation: [Mr., Ms., Mrs., Little]
- Current Hour: [16]

Please note, that our Glossary does not explicitly specify which variables are unknown (otherwise there will be one more column “Unknown” following the column “Domain”). So, Rule Solver™ automatically defines that only variables that are used inside decision table conclusions are unknown. Those variables
are: “AgeGroup”, “Greeting”, and “Salutation” and as you can see their domains continue to contain multiple possible values.

**Step 5.** Then Rule Solver™ posts (activates) constraints that correspond to the rules defined in our decision tables. The order of constraint posting really does not affect the final results. Rule Solver™ reports the state of our constrained variables after constraint posting:

```plaintext
=== After Posting Constraints:
Marital Status[Married]
Greeting[Good Afternoon]
Age[4]
Age Group[Little]
Gender[Female]
Salutation[Little]
Current Hour[16]
```

As you can see, in this simple case constraint posting was sufficient to instantiate all constraint variables (select single values from their domain).

**Step 6.** However, in many practical cases constraint posting is not sufficient and Rule Solver™ always executes its default search algorithm to find a single solution of an automatically created constraint satisfaction problem. Rule Solver™ also saves all found values of constrained variables back to the proper attributes of the business objects (based on which these variables were created).

**Simple Decision Model “DecisionLoan”**

In this example we will demonstrate a simple loan pre-qualification model that should decide whether to approve or decline a loan application.

**Decision Model**

We will use Excel to represent decisions, related decision tables, and several test cases. We will start with the main table of the type “Decision” that consists of 4 sub-decisions:
The first sub-decision “Calculate Internal Variables” will be implemented using the following decision table:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is More Than</td>
<td>$I{\text{Monthly Debt}} \times \frac{1}{2} \times \text{Loan Term}^2$</td>
</tr>
<tr>
<td>Is $\leq$</td>
<td>$I{\text{Total Debt}} \times \frac{1}{2} \times \text{Loan Term}^2$</td>
</tr>
</tbody>
</table>

The second sub-decision “Validate Income” will be implemented using the following decision table:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Income Is More Than</td>
<td>$I{\text{Monthly Income}} \times \text{Loan Term}$</td>
</tr>
<tr>
<td>Total Income Is $\leq$</td>
<td>$I{\text{Total Debt}} \times \text{Loan Term}$</td>
</tr>
</tbody>
</table>
And the fourth decision “Summarize” will be implemented using the following decision table:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Condition</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is</td>
<td>Unsufficient</td>
<td>Is NOT QUALIFIED</td>
</tr>
<tr>
<td>Is</td>
<td>Sufficient</td>
<td>Is Low</td>
</tr>
<tr>
<td>Is</td>
<td>Sufficient</td>
<td>Is One Of Mid, High</td>
</tr>
</tbody>
</table>

The proper Glossary for this model can be defined as follows:

<table>
<thead>
<tr>
<th>Decision Variable</th>
<th>Object</th>
<th>Attribute</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monthly Income</td>
<td>Customer</td>
<td>monthlyIncome</td>
<td>0-5000000</td>
</tr>
<tr>
<td>Mortgage Holder</td>
<td>mortgageHolder</td>
<td>Yes, No</td>
<td></td>
</tr>
<tr>
<td>Outside Credit Score</td>
<td>outsideCreditScore</td>
<td>0-999</td>
<td></td>
</tr>
<tr>
<td>Loan Holder</td>
<td>loanHolder</td>
<td>Yes, No</td>
<td></td>
</tr>
<tr>
<td>Credit Card Balance</td>
<td>creditCardBalance</td>
<td>-1000000 - 100000000</td>
<td></td>
</tr>
</tbody>
</table>
We will assume that the data for our model comes not from Java but rather from tables defined in Excel. The following table defines a datatype “Customer”:

<table>
<thead>
<tr>
<th>Datatype Customer</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>fullName</td>
</tr>
<tr>
<td>String</td>
<td>SSN</td>
</tr>
<tr>
<td>int</td>
<td>monthlyIncome</td>
</tr>
<tr>
<td>int</td>
<td>monthlyDebt</td>
</tr>
<tr>
<td>String</td>
<td>mortgageHolder</td>
</tr>
<tr>
<td>int</td>
<td>outsideCreditScore</td>
</tr>
<tr>
<td>String</td>
<td>loanHolder</td>
</tr>
<tr>
<td>int</td>
<td>creditCardBalance</td>
</tr>
<tr>
<td>int</td>
<td>educationLoanBalance</td>
</tr>
<tr>
<td>String</td>
<td>internalCreditRating</td>
</tr>
<tr>
<td>String</td>
<td>internalAnalystOpinion</td>
</tr>
</tbody>
</table>

The following table defines a datatype “LoanRequest”:

<table>
<thead>
<tr>
<th>Datatype LoanRequest</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>customer</td>
</tr>
<tr>
<td>int</td>
<td>amount</td>
</tr>
<tr>
<td>String</td>
<td>purpose</td>
</tr>
<tr>
<td>int</td>
<td>term</td>
</tr>
<tr>
<td>String</td>
<td>incomeValidationResult</td>
</tr>
<tr>
<td>String</td>
<td>debtResearchResult</td>
</tr>
<tr>
<td>String</td>
<td>loanQualificationResult</td>
</tr>
</tbody>
</table>
The following table defines a datatype “InternalVariables”:

<table>
<thead>
<tr>
<th>Datatype</th>
<th>InternalVariables</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>totalIncome</td>
</tr>
<tr>
<td>int</td>
<td>totalDebt</td>
</tr>
</tbody>
</table>

These internal variables can be created in the following table:

<table>
<thead>
<tr>
<th>Variable</th>
<th>InternalVariables internal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Income</td>
<td>Total Debt</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Initially they are unknown (we use zeros).

Now we may define test-requests using the following Data table:

<table>
<thead>
<tr>
<th>Data</th>
<th>LoanRequest loanRequests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>customer</td>
</tr>
<tr>
<td>Customer</td>
<td>Peter N. Johnson</td>
</tr>
<tr>
<td></td>
<td>Mary K. Brown</td>
</tr>
</tbody>
</table>

Thus, our data is specified and we should connect instances of our test-objects with business concepts specified on the glossary. It can be done using the following table of the type “DecisionObject”:

<table>
<thead>
<tr>
<th>DecisionObject decisionObjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Concept</td>
</tr>
<tr>
<td>Customer</td>
</tr>
<tr>
<td>Request</td>
</tr>
<tr>
<td>Internal</td>
</tr>
</tbody>
</table>

This completes the definition of our decision model. The next step is to find solutions for this model. We will start with a regular (sequential) OpenRules® rule engine that will execute our decision.
Solution with Rule Engine

The standard OpenRules® installation comes with the proper decision project “DecisionLoanCP” in the workspace “openrules.solver”. The main Java class Main.java contains one method “main” that creates and executes our Excel-based decision:

```java
public static void main(String[] args) {
    String fileName = "file:rules/main/Decision.xls";
    Decision decision =
        new Decision("DetermineLoanPreQualificationResults", fileName);
    decision.execute();
}
```

Note that here we omitted the statement

```java
System.setProperty("OPENRULES_MODE", "Execute");
```

because the mode “Execute” is used by default. After execution of this code we will receive the following results:

```plaintext
*** Decision DetermineLoanPreQualificationResults ***
Decision has been initialized
Decision DetermineLoanPreQualificationResults: Calculate Internal Variables
Conclusion: Total Debt Is 165600
Conclusion: Total Income Is 360000
Decision DetermineLoanPreQualificationResults: Validate Income
Decision DetermineLoanPreQualificationResults: Debt Research
Conclusion: Debt Research Result Is High
Decision DetermineLoanPreQualificationResults: Summarize
Conclusion: Loan Qualification Result Is QUALIFIED
Decision has been finalized
```

Please note that the variable “Loan Qualification Result” depends on the variables “Income Validation Result” and “Debt Research Result”. So, for the sequential rule engine it is extremely important that sub-decisions “Validate Income” and “Debt Research” are specified before the sub-decision “Summarize”.

Now we find a solution for the same decision model using OpenRules® Rule Solver™ that resolves rules sequencing issues automatically.
Solution with Rule Solver

The main Java class MainCP.java contains one method “main” that is similar to the previous one but contains one extra line before creation of the decision:

```java
public static void main(String[] args) {
    String fileName = "file:rules/main/Decision.xls";
    System.setProperty("OPENRULES_MODE", "Solve");
    Decision decision =
        new Decision("DetermineLoanPreQualificationResults", fileName);
    decision.execute();
}
```

The statement

```java
System.setProperty("OPENRULES_MODE", "Solve");
```

enforces the use of Rule Solver™ instead of the default (sequential) rule engine. After execution of this model with Rule Solver™ we will receive the following results:

**Step 1.**
*** Decision DetermineLoanPreQualificationResults ***
Create RuleSolver
JSR-331 Implementation based on Constrainer 5.4 (light)
=== Rule Solver (version 6.2.0) ===
addConstrainedVariables
Decision has been initialized with RuleSolver

**Step 2.**
=== Initial Problem Variables:
Outside Credit Score[0..999]
Education Loan Balance[-100000..100000000]
Monthly Income[0..5000000]
Credit Card Balance[-100000..100000000]
Income Validation Result[SUFFICIENT,UNSUFFICIENT,?]
Debt Research Result[High,Mid,Low,?]
Mortgage Holder[Yes,No]
Total Debt[0..500000]
Total Income[0..500000]
Loan Qualification Result[QUALIFIED,NOT QUALIFIED,?]
Loan Holder[Yes,No]
Internal Credit Rating[A,B,C,D,F]
Internal Analyst Opinion[High,Mid,Low]

**Step 3.**
Decision DetermineLoanPreQualificationResults: Calculate Internal Variables
Decision DetermineLoanPreQualificationResults: Validate Income
Decision DetermineLoanPreQualificationResults: Debt Research
Decision DetermineLoanPreQualificationResults: Summarize

**Step 4.**

--- After Assigning Data:
- Outside Credit Score[720]
- Education Loan Balance[0]
- Monthly Income[5000]
- Credit Card Balance[2500]
- Income Validation Result[SUFFICIENT, UNSUFFICIENT, ?]
- Debt Research Result[High, Mid, Low, ?]
- Mortgage Holder[Yes]
- Total Debt[0..500000]
- Total Income[0..500000]
- Loan Qualification Result[QUALIFIED, NOT QUALIFIED, ?]
- Loan Holder[No]
- Internal Credit Rating[A]
- Internal Analyst Opinion[Low]

**Step 5.**

--- After Posting Constraints:
- Outside Credit Score[720]
- Education Loan Balance[0]
- Monthly Income[5000]
- Credit Card Balance[2500]
- Income Validation Result[SUFFICIENT]
- Debt Research Result[High]
- Mortgage Holder[Yes]
- Total Debt[165600]
- Total Income[360000]
- **Loan Qualification Result[QUALIFIED]**
- Loan Holder[No]
- Internal Credit Rating[A]
- Internal Analyst Opinion[Low]

**Step 6.**

--- Solve ---

The descriptions of the “green steps” are similar to the ones described in the previous example. As you can see, after all execution steps (described in the previous example) we received the same results as those produced by the rule engine. The rule engine produced

**Conclusion:** Loan Qualification Result Is QUALIFIED

while **Rule Solver™** instantiated the variable “Loan Qualification Result” with the value “QUALIFIED”:

**Loan Qualification Result[QUALIFIED]**
Here again pure constraint propagation was sufficient to find one (and only one) solution of this problem. However, contrary to the rule engine, now we can freely change the order of sub-decisions in the main table “Decision”.

**Decision Modeling with Rules Solver**

Previous examples demonstrate how Rule Solver™ may be used instead of the standard rule engine. For the same Excel-based decision model Rule Solver™ executes all related business rules and either infers a decision or diagnoses conflicts among rules and input data.

**Rule Solver™ Benefits**

Rule Solver™ brings several important benefits to decision modeling:

- No explicit ordering between decision tables
- No explicit ordering of rules within decision tables
- Automatic validation of conflicts between rules across all decision tables
- Automatic check of decision models for completeness and an ability to find a decision when rules do not cover all possible combinations of decision variables.

**How Rule Solver™ Works**

Formally, we may describe a decision model as follows:

- There is a set of business objects $X = \{ X_1, ..., X_n \}$
- Each business object $X_i$ contains decision variables $V_i = \{ V_1, ..., V_m \}$ with possible values $D_j = \{ v_{j1}, ..., v_{jk} \}$ for each variable $V_j$
- There is a set of rules $R = \{ R_1, ..., R_r \}$, where a rule $R_k$ defines relationships between different decision variables by specifying the allowed combinations for all variables in that rule.
The rules from set R are grouped into rule sets usually called Decision Tables. Execution of a decision model should cause the assignment of values to all decision variables that satisfy all rules.

To execute a decision model Rule Solver™ does the following:

1) Reads a decision model created by business analysts directly from the rule repository (usually from a set of Excel files)

2) Generates a constraint satisfaction problem (CSP) using JSR-331 CP API:
   - Creates a CSP instance
   - Creates constrained variables for all unknown decision variables described in the decision model’s glossary
   - Creates constraints that correspond to the rules defined in all decision tables

3) Validate the consistency of the model by checking the consistency of the generated CSP and points to possible conflicts using the business terms of the initial decision model

4) Executes the decision model against concrete data using the following steps:
   - instantiating all constrained variables for which input data is defined
   - posting all constraints that correspond to rules from all decision tables
   - if constraint propagation by itself does not find single values for all decision variables (does not instantiate all constrained variables), then runs a constraint solver’s search strategy that finds a solution of the CSP.

Generating CSP

Rule Solver™ creates a CSP instance during initialization of the decision model. It is done by the standard method “customInitializeDecision” within the file “DecisionTemplateSolveTempalte.xls”. The actual CSP is an instance of the predefined class RuleSolver created by the following method:

```java
RuleSolverFactory.newRuleSolver("solver",decision.getEngine())
```

Then Rule Solver™ adds constrained variables using the following method:
This method iterates through the decision model's glossary and for each decision variable creates a constrained variable of one of the following types:

- Var for integer constrained variables
- VarString for string constrained variables
- VarBool for Boolean constrained variables
- VarReal for real constrained variables
- VarSet for set constrained variables.

*Note.* The current version is limited to only two types: Var and VarString.

### Adding Constrained Variables

Rule Solver™ automatically converts decision variable domains from the glossary to the domains of the constrained variables as they are specified by JSR 331. While the glossary does not specify a particular type of the decision variables, the concrete types of constrained variables are defined based on the provided data instances. For example, a constrained variable that corresponds to the decision variable "Monthly Income" will be created using the following JSR 331 method:

```
rs.variable("Monthly Income", 0, 50000000);
```

### Adding Constraints

During processing the tables of the type "Decision" Rule Solver™ executes all decision tables and for every rule adds a conditional constraint that has the following form:

```
conditionConstraints.implies(conclusionConstraint)
```

Here “conditionConstraints” are accumulated by using the method “and” defined for the JSR-331 class Constraint. For example, the rule

```
IF Person Years at Current Employer < 1
```
AND Person Number of Jobs in Past Five Years > 5

THEN Person Employment History = Poor

may be implemented in Java using the JSR-331 interface:

Var var1 = rs.getVar("Person Years at Current Employer");
Constraint c1 = rs.linear(var1, "<", 1);

Var var2 = rs.getVar("Person Number of Jobs in Past Five Years");
Constraint c2 = rs.linear(var2, ">", 5);

Constraint conditionConstraints = c1.and(c2);

VarString var3 = rs.getVarString("Person Employment History");
Constraint conclusionConstraint = rs.linear(var3, ",=", "Poor");
rs.add(conditionConstraints.implies(conclusionConstraint));

This way Rule Solver™ creates all constraints but it does not post (activate) them yet.

Posting Data Constraints

Then Rule Solver™ assigns data to decision variables that are already known. How does Rule Solver™ differentiate between known and unknown decision variables? A glossary may explicitly specify which variables are unknown in the column “Unknown” that follows the column “Domain”. If this optional column is not used then Rule Solver™ automatically defines them as those variables that are used inside decision table conclusions only. Rule Solver™ reports the state of all (known and unknown) constrained variables

Posting Problem Constraints

After posting data constraint, Rule Solver™ posts (activates) problem constraints that correspond to all rules defined in all decision tables. The order of constraint posting is not important as it does not affect the final results. Rule Solver™ reports the state of all (known and unknown) constrained variables after constraint posting. Because of constraint propagation (that may depend on the
applied underlying CP solver) a combination of data and problem constraints may instantiate all variables or it may diagnose possible conflicts. In many cases constraint posting is sufficient to instantiate all constraint variables (select single values from their domain).

Solving the Problem

In some cases constraint posting is not sufficient to instantiate all constraint variables. That’s why Rule Solver™ always executes its default search algorithm to find a single solution of an automatically created constraint satisfaction problem. Rule Solver™ also saves all found values of constrained variables back to the proper attributes of the business objects (based on which these variables were created). A user may also control how to solve the problems choosing one of 3 options:

1) **Find a solution**: the is the default option

2) **Find all solutions**: the solver is trying to find as many different solutions as possible but no more the defined by the limit used in the statement

    decision.put("MaxSolutions",maxNumberOfSolutions);

3) **Find an optimal solution**: the solver is trying to find a solution that minimizes an objective variable specified in the statement

    decision.put("Minimize",nameOfCostVariable);

To maximize the cost variable use the statement

    decision.put("Maximize",nameOfCostVariable);

During the optimization process Rule Solver™ considers no more solutions than a number specified in the statement

    decision.put("MaxSolutions",maxNumberOfSolutions);

See example of all 3 problem solving modes in the example below.
Using Templates

Rule Solver™ never generates Java code. Instead, at run-time, it simply creates an instance of different JSR-331 classes and adds them to the already created constraint satisfaction problem. All instances of constrained variables and constraints are added to the problem “on the fly”. To do that, Rule Solver™ effectively utilizes the existing OpenRules® templatization mechanism. OpenRules® uses different rule templates to implement all tables included into the default (not constraint-based) implementation of the decision model. Such tables as “Decision”, “DecisionTable”, and “Glossary” are actually implemented based on rule templates defined in several configuration Excel files. For example, the file “DecisionTableExecuteTemplates.xls” contains a template with the fixed name “DecisionTableTemplate” and all Decision Tables are created based on it. This template is a regular OpenRules “single-hit” rules table. It means that it is trying to execute rules in top-down order by evaluating their conditions. When all conditions inside a rule are evaluated as TRUE, the rule’s conclusion (and possibly other related actions) will be executed and all remaining rules will be ignored.

Rule Solver™ provides the configuration file “DecisionTableSolveTemplates.xls” that substitutes the template “DecisionTableTemplate” with a different implementation that is actually a special “RuleSequence” decision table. This table unconditionally executes all (!) rules inside every decision table one after another. However, instead of evaluating rule conditions it simply creates new constraints similar to $c_1$ and $c_2$ above, and then “AND”s all previously defined conditions similarly to $c_1$.and($c_2$). Thus, all conditions from one rule will form a constraint conditionConstraints described in the previous example. Then the conclusion will be converted to the conclusionConstraint that is based on the constrained variable associated with the conclusion’s fact type, operator, and value. Finally, Rule Solver™ creates a new constraint conditionConstraints.implies(conclusionConstraint) and adds it to the problem. According to the JSR-331, this constraint states that if the
constraint conditionConstraint is satisfied then the constraint conclusionConstraint also should be satisfied.

While the “DecisionTableTemplate” may contain more complicated constructions, the very fact that the generated CSP can be reconfigured by simply changing the template directly in Excel, makes this approach extremely flexible, extensible, and customizable for different needs.

Using JSR-331

The use of the standard JSR 331 allows a user not to commit to a particular CP solver. A user may try different underlying solvers with the same decision model before choosing the most suitable one based on its technical and business applicability. A user may switch between different underlying CP solvers compliant with the JSR 331 without any changes in the code.

Decision Execution Reports

OpenRules® provides an ability to generate decision execution reports in the HTML-format. To generate an execution report, you should add the following setting to the decision’s Java launcher:

```java
decision.put("report", "On");
```

before calling `decision.execute()`. By default, execution reports are not generated as they are needed mainly for decision analysis. Reports are regenerated for every decision run.

When you use the OPENRULES_MODE equal to “Solve”, Rule Solver™ generates an execution report. The name of the generated report is formed using the name of the proper decision with an extension “.html”. By default, this report is placed in the folder "report" of the main decision folder. If you want to change the report path say to "c:/temp/report" you may achieve it by stating:
decision.put("report","On");
decision.put("reportPath","c:/temp/report");

If the name of the decision was “ScheduleActivitiesWithAlternativeResources”, then the generated report will be placed to the file "c:/temp/report/Report.ScheduleActivitiesWithAlternativeResources.html".

If you want to also use a custom report’s name instead of the decision’s name, say "MyReport", you may add one more line:

decision.put("reportName","MyReport");

The report will be placed to "c:/temp/report/MyReport.html".

If the folder “report” does not exist in “c:/temp/”, it will be automatically created. Below is an example of the execution report for the sample project “DecisionLoan”:

**OpenRules Solver Execution Report**

**Decision "DetermineLoanPreQualificationResults"**

**Wed Apr 23 16:41:58 EDT 2014**

--- Assigning Data To Decision Variables ---

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Initial State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outside Credit Score</td>
<td>620</td>
<td>Known</td>
</tr>
<tr>
<td>Loan Term</td>
<td>72</td>
<td>Known</td>
</tr>
<tr>
<td>Education Loan Balance</td>
<td>23800</td>
<td>Known</td>
</tr>
<tr>
<td>Monthly Income</td>
<td>5000</td>
<td>Known</td>
</tr>
<tr>
<td>Credit Card Balance</td>
<td>5654</td>
<td>Known</td>
</tr>
<tr>
<td>Income Validation Result</td>
<td>SUFFICIENT, UNSUFFICIENT, ?</td>
<td>Unknown</td>
</tr>
</tbody>
</table>
### Debt Research Result

<table>
<thead>
<tr>
<th>Debt Research Result</th>
<th>High, Mid, Low, ?</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total Debt</strong></td>
<td>0..500000</td>
<td>Unknown</td>
</tr>
<tr>
<td><strong>Mortgage Holder</strong></td>
<td>No</td>
<td>Known</td>
</tr>
<tr>
<td><strong>Total Income</strong></td>
<td>0..500000</td>
<td>Unknown</td>
</tr>
<tr>
<td><strong>Monthly Debt</strong></td>
<td>2300</td>
<td>Known</td>
</tr>
<tr>
<td><strong>Loan Qualification Result</strong></td>
<td>QUALIFIED, NOT QUALIFIED, ?</td>
<td>Unknown</td>
</tr>
<tr>
<td><strong>Loan Holder</strong></td>
<td>No</td>
<td>Known</td>
</tr>
<tr>
<td><strong>Internal Credit Rating</strong></td>
<td>B</td>
<td>Known</td>
</tr>
<tr>
<td><strong>Internal Analyst Opinion</strong></td>
<td>Low</td>
<td>Known</td>
</tr>
</tbody>
</table>

---

### Posting If-Then Constraints

#### Posted If-Then Constraints

<table>
<thead>
<tr>
<th>Status</th>
<th>Constraints</th>
<th>Decision Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>On</td>
<td>Total Debt = 216000</td>
<td>CalculateInternalVariables</td>
</tr>
<tr>
<td>On</td>
<td>Total Income = 460800</td>
<td>CalculateInternalVariables</td>
</tr>
<tr>
<td>On</td>
<td>IF Total Income &gt; 403200 THEN Income Validation Result = SUFFICIENT</td>
<td>DetermineIncomeValidationResult</td>
</tr>
<tr>
<td>On</td>
<td>IF Total Income &lt;= 403200 THEN Income Validation Result = UNSUFFICIENT</td>
<td>DetermineIncomeValidationResult</td>
</tr>
<tr>
<td>On</td>
<td>IF Mortgage Holder = Yes THEN Debt Research Result = High</td>
<td>DetermineDebtResearchResult</td>
</tr>
<tr>
<td>On</td>
<td>IF Mortgage Holder = No AND Outside Credit Score &gt; 100 AND Outside Credit Score &lt;= 550 THEN Debt Research Result = High</td>
<td>DetermineDebtResearchResult</td>
</tr>
<tr>
<td>On</td>
<td>IF Mortgage Holder = No AND Outside Credit Score &gt; 550 AND Outside Credit Score &lt;= 900 AND Loan Holder = Yes AND Credit Card</td>
<td>DetermineDebtResearchResult</td>
</tr>
<tr>
<td>Condition</td>
<td>Action</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>---------------------------------</td>
<td></td>
</tr>
<tr>
<td>Balance &lt;= 0 THEN Debt Research Result = Mid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>On</td>
<td>DetermineDebtResearchResult</td>
<td></td>
</tr>
<tr>
<td>IF Mortgage Holder = No AND Outside Credit Score &gt; 550 AND Outside Credit Score &lt;= 900 AND Loan Holder = Yes AND Credit Card Balance &gt; 0 AND Education Loan Balance &gt; 0 THEN Debt Research Result = High</td>
<td></td>
<td></td>
</tr>
<tr>
<td>On</td>
<td>DetermineDebtResearchResult</td>
<td></td>
</tr>
<tr>
<td>IF Mortgage Holder = No AND Outside Credit Score &gt; 550 AND Outside Credit Score &lt;= 900 AND Loan Holder = Yes AND Credit Card Balance &gt; 0 AND Education Loan Balance &lt;= 0 AND Internal Credit Rating ISONEOF D, F THEN Debt Research Result = Mid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>On</td>
<td>DetermineDebtResearchResult</td>
<td></td>
</tr>
<tr>
<td>IF Mortgage Holder = No AND Outside Credit Score &gt; 550 AND Outside Credit Score &lt;= 900 AND Loan Holder = No AND Credit Card Balance &lt;= 0 AND Education Loan Balance &lt;= 0 THEN Debt Research Result = Low</td>
<td></td>
<td></td>
</tr>
<tr>
<td>On</td>
<td>DetermineDebtResearchResult</td>
<td></td>
</tr>
<tr>
<td>IF Mortgage Holder = No AND Outside Credit Score &gt; 550 AND Outside Credit Score &lt;= 900 AND Loan Holder = No AND Credit Card Balance &lt;= 0 AND Education Loan Balance &gt; 0 AND Internal Credit Rating ISONEOF D, F THEN Debt Research Result = High</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
AND Internal Credit Rating ISONEOF A, B, C THEN Debt Research Result = Low

On
IF Mortgage Holder = No AND Internal Analyst Opinion = High THEN Debt Research Result = High
DetermineDebtResearchResult

On
IF Mortgage Holder = No AND Internal Analyst Opinion = Mid THEN Debt Research Result = Mid
DetermineDebtResearchResult

On
IF Mortgage Holder = No AND Internal Analyst Opinion = Low THEN Debt Research Result = Low
DetermineDebtResearchResult

On
IF Income Validation Result = UNSUFFICIENT THEN Loan Qualification Result = NOT QUALIFIED
DetermineLoanQualificationResult

On
IF Income Validation Result = SUFFICIENT AND Debt Research Result = Low THEN Loan Qualification Result = NOT QUALIFIED
DetermineLoanQualificationResult

On
IF Income Validation Result = SUFFICIENT AND Debt Research Result ISONEOF Mid, High THEN Loan Qualification Result = QUALIFIED
DetermineLoanQualificationResult

---

**Solution #1**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outside Credit Score</td>
<td>735</td>
</tr>
<tr>
<td>Loan Term</td>
<td>72</td>
</tr>
<tr>
<td>Education Loan Balance</td>
<td>0</td>
</tr>
<tr>
<td>Monthly Income</td>
<td>6400</td>
</tr>
<tr>
<td>Credit Card Balance</td>
<td>1200</td>
</tr>
<tr>
<td>Income Validation Result</td>
<td>SUFFICIENT</td>
</tr>
<tr>
<td>Debt Research Result</td>
<td>High</td>
</tr>
<tr>
<td>Total Debt</td>
<td>216000</td>
</tr>
<tr>
<td>Mortgage Holder</td>
<td>Yes</td>
</tr>
<tr>
<td>Total Income</td>
<td>460800</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Loan Qualification Result</td>
<td>QUALIFIED</td>
</tr>
<tr>
<td>Monthly Debt</td>
<td>3000</td>
</tr>
<tr>
<td>Loan Holder</td>
<td>Yes</td>
</tr>
<tr>
<td>Internal Credit Rating</td>
<td>C</td>
</tr>
<tr>
<td>Internal Analyst Opinion</td>
<td>Mid</td>
</tr>
</tbody>
</table>

If some errors occur during the constraint posting they will be shown in red.

**Decision Output**

Actually, when you execute a decision using a constraint-based inference engine it always returns solution(s) as a decision's output. When your decision is trying to find one feasible or an optimal solution, then after calling Decision's method `decision.execute()` you may write

```java
Solution solution = (Solution) decision.getOutput();
```

If the "solution" is null, it means no solution is found. Otherwise, you may print a solution using `solution.log()` or analyze it using the JSR331 Solution API.

If your decision is trying to find many feasible solutions, then after `decision.execute()` you may write

```java
Solution[] solution = (Solution[]) decision.getOutput();
```

and to analyze the array of the produced solutions.

**Implementation Restrictions and Future Improvements**

1) Rule Solver™ only supports decision tables of the type “DecisionTable” (and not “DecisionTable1” or “DecisionTable2”).

2) The current release only works with integer and string decision variables of the type. Boolean, real, and set variables will be supported in the next releases.
Rule Solver™ can be used as a business-oriented constraint solver. It provides an ability to represent and solve constraint satisfaction and optimization problems using Excel-based decision tables oriented to business users.

Usually constraint solver requires a software expert familiar with a particular CP language or API. Contrary, Rule Solver™ allows a non-technical user to define a constraint satisfaction problem using Excel-based business rules (decision tables) without becoming a CP guru. Rule Solver™ supports declarative application development concentrating on WHAT TO DO (problem definition) instead of HOW TO DO it (problem resolution). After a constraint satisfaction problem is specified in Excel-based decision tables, it will be automatically solved by an underlying constraint solver.

Being based on the standard JSR 331 “Constraint Programming API” defined by the Java Community Process, Rule Solver™ allows a user to switch between different JSR-331 compliant solvers without any changes in the application code.

Constraint Satisfaction Problem (CSP)

Many real-life problems that deal with multiple alternatives with many unknowns subject to different constraints can be presented and solved as constraint satisfaction problems (CSP).

Formally a constraint satisfaction problem is defined by

- a set of **variables** $V_1, V_2, \ldots, V_n$, and
- a set of **constraints** $C_1, C_2, \ldots, C_m$. 
Each variable $V_i$ has a non-empty domain $D_i$ of possible values. Each constraint $C_j$ involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all of the variables. A solution to a CSP is an assignment that satisfies all the constraints. If a CSP requires a solution that maximizes or minimizes an objective function it is called a “constraint optimization problem”. We will use the abbreviation CSP for both types of problems.

The main CSP search technique interleaves various forms of search with constraint propagation, in which infeasible values are removed from the domains of the variables through reasoning about the constraints.

**Major CP Concepts**

CP supports a clear separation between Problem Definition and Problem Resolution. At the very high level a business user that defines a CSP is presented with only 6 major concepts:

- **Problem** (for problem definition)
  - Constrained Variable
  - Constraint

- **Solver** (for problem resolution)
  - Search Strategy
  - Solution

While different constraint solvers use diverse names and representations for CP concepts, semantically these 6 concepts are invariants for the majority of solvers. JSR-331 provides a unified naming convention and detailed specifications for these concepts. You may want to read the JSR-331 User Manual which contains a variety of Java examples.

Below we explain how to represent and solve different CSPs using MS Excel™ and OpenRules® decision tables. Contrary to Java, Rule Solver™ allows subject matter experts (non-programmers) to utilize the power of constraint
programming. Rule Solver™ provides a set of templates (defined in Excel) that allow a user to define and solve different CSPs using only simple Excel-based decision tables. The following example will demonstrate how to create decision models for simple CSPs.

**Introductory Example “SEND+MORE=MONEY”**

This example demonstrates how to represent and solve a simple puzzle using Rule Solver™. Assuming that different letters represent different digits, you need to solve the following puzzle:

```
  S E N D
+  M O R E
==========
M O N E Y
```

**Excel-based Decision Model**

The standard Rule Solver™ installation comes with the proper decision project “DecisionSendMoreMoney” in the workspace “openrules.solver”. This project has only one file “Decision.xls” that includes all needed Excel tables.

Our problem has only 8 different decision variables S, E, N, D, M, O, R, and Y. They are integer variables with domains from 0 to 9. Variables S and M have a possible minimal value of 1 because they at the beginning of the words SEND and MORE. So, first we will define a glossary:

<table>
<thead>
<tr>
<th>Fact Name</th>
<th>Business Concept</th>
<th>Attribute</th>
<th>Domain</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Puzzle</td>
<td>s</td>
<td>1-9</td>
<td>TRUE</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>e</td>
<td>0-9</td>
<td>TRUE</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td>n</td>
<td>0-9</td>
<td>TRUE</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>d</td>
<td>0-9</td>
<td>TRUE</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td>m</td>
<td>1-9</td>
<td>TRUE</td>
</tr>
<tr>
<td>O</td>
<td></td>
<td>o</td>
<td>0-9</td>
<td>TRUE</td>
</tr>
<tr>
<td>R</td>
<td></td>
<td>r</td>
<td>0-9</td>
<td>TRUE</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>y</td>
<td>0-9</td>
<td>TRUE</td>
</tr>
<tr>
<td>SEND</td>
<td></td>
<td>send</td>
<td>1000-9999</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
We added to the end of the glossary intermediate variables SEND, MORE, and MONEY that will be used to simplify the expression of the main problem constraint. All these variables are unknown. When we use a glossary we need to create actual data instances (business objects). In this case we will simple use the proper Excel tables:

<table>
<thead>
<tr>
<th>MORE</th>
<th>money</th>
<th>1000-9999</th>
<th>TRUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONEY</td>
<td>10000-99999</td>
<td>TRUE</td>
<td></td>
</tr>
</tbody>
</table>

The initial values 0 will be ignored as all our variables are unknown. To connect the data objects with the glossary we need the following table:

<table>
<thead>
<tr>
<th>DecisionObject decisionObjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Concept</td>
</tr>
<tr>
<td>Business Object</td>
</tr>
</tbody>
</table>

| Puzzle := puzzle |

Now we need to define our intermediate variables SEND, MORE, and MONEY as scalar products using the following table:
The first row of this decision table defines SEND as a scalar product \(S\times1000+E\times100+N\times10+1\times D\). Similarly the second and third rows define variables MORE and MONEY using the proper scalar products.

Now we may define the main problem constraint SEND + MORE = MONEY:

To state that our 8 main decision variables are different we may use a decision table for the AllDiff constraint:

This completes our decision model. To print the result directly from Excel, we may add one more method “PrintSolution”:
To execute this model with Rule Solver™ we will use a Java class `Main.java` with the following main-method:

```java
public static void main(String[] args) {
    String fileName = "file:rules/Decision.xls";
    System.setProperty("OPENRULES_MODE", "Solve");
    Decision decision = new Decision("SendMoreMoney",fileName);
    decision.put("trace","On");
    decision.execute();
    decision.execute("PrintSolution",decision);
}
```

Here are the execution results:

```plaintext
*** Decision SendMoreMoney ***
Create RuleSolver
JSR-331 Implementation based on Constrainer 5.4 (light)
=== Rule Solver (version 6.2.0) ===
addConstrainedVariables
Decision has been initialized with RuleSolver
=== Initial Problem Variables:
D[0..9]
E[0..9]
SEND[1000..9999]
MONEY[10000..99999]
S[1..9]
MORE[1000..9999]
R[0..9]
M[1..9]
N[0..9]
O[0..9]
Y[0..9]
Decision SendMoreMoney: Add Intermediate Variables
Decision SendMoreMoney: Main Constraint
Decision SendMoreMoney: All Different
=== After Assigning Data:
D[0..9]
E[0..9]
SEND[1000..9999]
```

```java
Method void PrintSolution(Decision decision)
System.out.println("
"n=====" +
"\n " + $\{SEND\} +
"\n + $\{MORE\} +
"\n=====" +
"\n " + $\{MONEY\} +
"\n=====");
```
MONEY[10000..99999]
S[1..9]
MORE[1000..99999]
R[0..9]
M[1..9]
N[0..9]
O[0..9]
Y[0..9]

=== After Posting Constraints:
D[0..9]
E[0..9]
SEND[8801..9999]
MONEY[10000..11198]
S[8..9]
MORE[1000..1199]
R[0..9]
M[1]
N[0..9]
O[0..1]
Y[0..9]

=== Solve ===

======
9567
+1085
======
10652
======

Pure Java Solution

You may want to compare the decision model with a pure Java solution - the proper JSR-331 code will look as follows:

```java
package org.jcp.jsr331.samples;
import javax.constraints.Problem;
import javax.constraints.ProblemFactory;
import javax.constraints.Var;
public class SendMoreMoney {
    public static void main(String[] args) {
        Problem p = ProblemFactory.newProblem("SendMoreMoney");
        // define variables
        Var S = p.variable("S",1,9);
        Var E = p.variable("E",0,9);
        Var N = p.variable("N",0,9);
        Var D = p.variable("D",0,9);
        Var M = p.variable("M",1,9);
        Var O = p.variable("O",0,9);
        Var R = p.variable("R",0,9);
```
Var Y = p.variable("Y", 0, 9);

// Post "all different" constraint
Var[] vars = new Var[] { S, E, N, D, M, O, R, Y }; p.postAllDiff(vars);

// Define expression SEND
int coef1[] = { 1000, 100, 10, 1 ];
Var[] sendVars = { S, E, N, D }; Var SEND = p.scalProd(coef1, sendVars);
SEND.setName("SEND");

// Define expression MORE
Var[] moreVars = { M, O, R, E }; Var MORE = p.scalProd(coef1, moreVars);
MORE.setName("MORE");

// Define expression MONEY
Var[] moneyVars = { M, O, N, E, Y }; int coef2[] = { 10000, 1000, 100, 10, 1 }; Var MONEY = p.scalProd(coef2, moneyVars);
MONEY.setName("MONEY"); p.add(MONEY);

// Post constraint SEND + MORE = MONEY
p.post(SEND.plus(MORE), "=" , MONEY);

// Problem Resolution
p.getSolver().findSolution();
p.log("Solution: " + SEND + " + " + MORE + " = " + MONEY);

This code will produce:

Solution: SEND[9567] + MORE[1085] = MONEY[10652]

Based on the levels of expertise of your users, you may decide what information to keep in Excel making it available to business users and what information to hard-code in Java.

Solving Arithmetic Problems

Let’s consider a simple arithmetic problem. There are four integer variables X, Y, Z, and R that may take values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, or 10. Considering that all variables should have different values, find a solution that satisfies the following constraints:
\[ X < Y \\
X + Y = Z \\
Z > 5. \]

Let's create a decision model for this problem. We will assume that the data for this problem is defined in a Java class XYZ:

```java
public class XYZ {
    int x, y, z;

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    public int getZ() {
        return z;
    }

    public void setZ(int z) {
        this.z = z;
    }

    public String toString() {
        return "x=" + x + " y=" + y + " z=" + z;
    }
}
```

The main-method to execute the decision model will look like below:

```java
public static void main(String[] args) {
    String fileName = "file:rules/main/Decision.xls";
    System.setProperty("OPENRULES_MODE", "Solve");
    Decision decision = new Decision("FindXYZ", fileName);

    XYZ xyz = new XYZ();
    decision.put("xyz", xyz);
    decision.put("trace", "On");
    decision.execute();
    out.println("\nDecision: " + xyz);
}
```

Let's define a glossary:
We may map the business concept “XYZ” to the actual object of the type XYZ using this DecisionObject table:

<table>
<thead>
<tr>
<th>Business Concept</th>
<th>Attribute</th>
<th>Domain</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>XYZ</td>
<td>x</td>
<td>0-10</td>
<td>TRUE</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>0-10</td>
<td>TRUE</td>
</tr>
<tr>
<td></td>
<td>z</td>
<td>0-10</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Our decision “FindXYZ” can be described in this table:

<table>
<thead>
<tr>
<th>Decisions</th>
<th>Execute Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define Var Z</td>
<td>XplusYeqZ</td>
</tr>
<tr>
<td>Binary Constraints</td>
<td>BinaryConstraints</td>
</tr>
</tbody>
</table>

And here are the proper decision tables for the problem constraints:

<table>
<thead>
<tr>
<th>ActionXoperYcompareZ</th>
<th>X &lt;oper&gt; Y &lt;compare&gt; Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X &lt; oper &gt; Y &lt; compare &gt; Z</td>
</tr>
<tr>
<td>X</td>
<td>+</td>
</tr>
<tr>
<td>Y</td>
<td>=</td>
</tr>
<tr>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DecisionTable XplusYeqZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionXoperYcompareZ</td>
</tr>
<tr>
<td>X &lt; oper &gt; Y &lt; compare &gt; Z</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>+</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>=</td>
</tr>
<tr>
<td>Z</td>
</tr>
</tbody>
</table>
Hopefully, these tables are self-explanatory. All these tables extend decision templates predefined in Rule Solver™ file “DecisionTableSolveTemplates.xls” included in the standard installation.

Find One Solution

After the execution of the above main-method we will receive these results:

```plaintext
*** Decision FindXYZ ***
Create RuleSolver
JSR-331 Implementation based on Constrainer 5.4 (light)
=== Rule Solver (version 6.2.0) ===
addConstrainedVariables
Decision has been initialized with RuleSolver
=== Initial Problem Variables:
    Y[0..10]
    X[0..10]
    Z[0..10]
Decision FindXYZ: Define Var Z
Decision FindXYZ: Binary Constraints
=== After Assigning Data:
    Y[0..10]
    X[0..10]
    Z[0..10]
=== After Posting Constraints:
    Y[1..10]
    X[0..9]
    Z[6..10]
=== Solve ===
    Decision: x=2 y=4 z=6
```

Find All Solutions

If you want to find not one but many possible solutions of this problem you may add the following statement to your Main.java program just before

```java
decision.execute();
```

Then Rule Solver™ will produce 21 different solutions:

```plaintext
=== Find All Solutions (but not more than 30)
```
Solution #1:
- Y[4]
- X[2]
- Z[6]

Solution #2:
- Y[4]
- X[3]
- Z[7]

Solution #3:
- Y[5]
- X[1]
- Z[6]

Find Optimal Solution

Now let’s add a new decision variable “Cost” defined in the domain [5-20] and equals to Cost = X*Y – Z. First we will add an “int” variable “cost” to our Java bean “XYZ”. Then we will modify the Glossary:

<table>
<thead>
<tr>
<th>Decision Variable</th>
<th>Business Concept</th>
<th>Attribute</th>
<th>Domain</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>XYZ</td>
<td>x</td>
<td>0-10</td>
<td>TRUE</td>
</tr>
<tr>
<td>Y</td>
<td>XYZ</td>
<td>y</td>
<td>0-10</td>
<td>TRUE</td>
</tr>
<tr>
<td>Z</td>
<td>XYZ</td>
<td>z</td>
<td>0-10</td>
<td>TRUE</td>
</tr>
<tr>
<td>Cost</td>
<td></td>
<td>cost</td>
<td>5-20</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

We will define a formula for the Cost in the following table:

<table>
<thead>
<tr>
<th>Decision Table DefineCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConclusionVar</td>
</tr>
<tr>
<td>Cost</td>
</tr>
<tr>
<td>Is $V[X].multiply ($V[Y] ) minus( $V[Z] )</td>
</tr>
</tbody>
</table>

Finally, we will add these formulas to the decision:

<table>
<thead>
<tr>
<th>Decision</th>
<th>FindXYZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decisions</td>
<td>Execute Rules</td>
</tr>
<tr>
<td>Define Var Z</td>
<td>XplusYeqZ</td>
</tr>
<tr>
<td>Binary Constraints</td>
<td>BinaryConstraints</td>
</tr>
<tr>
<td>Define Cost</td>
<td>DefineCost</td>
</tr>
</tbody>
</table>
If we try to run the same main program now (with “MaxSolutions” being set to 30) then Rule Solver™ will produce only 8 different solutions:

```
=== Find All Solutions (but not more than 30)
Solution #1:
  Cost[5]
  Y[4]
  X[3]
  Z[7]
Solution #2:
  Cost[5]
  Y[7]
  X[2]
  Z[9]
Solution #3:
  Cost[6]
  Y[8]
  X[2]
  Z[10]
Solution #4:
  Cost[7]
  Y[5]
  X[3]
  Z[8]
Solution #5:
  Cost[9]
  Y[6]
  X[3]
  Z[9]
Solution #6:
  Cost[11]
  Y[5]
  X[4]
  Z[9]
Solution #7:
  Cost[11]
  Y[7]
  X[3]
  Z[10]
Solution #8:
  Cost[14]
  Y[6]
  X[4]
  Z[10]
```

We may want to find a solution that maximizes the Cost. To do that we will add the following statement to your Main.java:

```
decision.put("Maximize", "Cost");
```

Rule Solver™ will produce:

```
Found a solution with [-5]
Found a solution with [-6]
```
Found a solution with [-7]
Found a solution with [-9]
Found a solution with [-11]
Found a solution with [-14]

*** Execution Profile ***
Number of Choice Points: 55
Number of Failures: 44
Occupied memory: 4842080
Execution time: 49 msec
Solution #6:

The proper decision project “DecisionXYZ” can be found in the standard Rule Solver™ installation in the workspace “openrules.solver”.

Sudoku Problem

The objective of this very popular game is to fill a 9×9 grid so that each column, each row, and each of the nine 3×3 boxes (also called blocks) contain the digits from 1 to 9, only one time each.

The standard Rule Solver™ installation comes with the proper decision project “DecisionSudoku” in the workspace “openrules.solver”. This project has only one file “Decision.xls” that includes all needed Excel tables.

We will start with the main table “Decision” (and will not use a glossary at all):

<table>
<thead>
<tr>
<th>Decision DefineAndSolveSudoku</th>
<th>Execute Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create 9x9 Matrix</td>
<td>:= solver(decision).variableMatrix(&quot;x&quot;,1,9,9,9)</td>
</tr>
<tr>
<td>Enter Known Problem Data</td>
<td>EnterSudokuData</td>
</tr>
<tr>
<td>Define Main Constraints</td>
<td>DefineSudokuConstraints</td>
</tr>
</tbody>
</table>

The very first sub-decision creates a 9x9 matrix “x” of constrained integer variables defined on the domain [1,9]. The element in the row i and column j has name “xij”.
We will enter Sudoku data using this Excel method:

```java
Method boolean EnterSudokuData(Decision decision)
{
    RuleSolver solver = decision.get("solver");
    VarMatrix matrix = solver.getVarMatrix("x");
    int[][] data = MatrixInt.getMatrix(problem1);
    for(int i = 0; i < 9; i++) {
        for(int j = 0; j < 9; j++)
            if (data[i][j] != 0)
                matrix.post(i, j, data[i][j]);
    }
}
```

that uses the following Data table:

```
<table>
<thead>
<tr>
<th></th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

To post all Sudoku constraints on this matrix we will use the following decision table:

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Define</td>
<td>Sudoku</td>
<td>Constraints</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>AllDiff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Variables</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x00, x01, x02, x03, x04, x05, x06, x07, x08</td>
<td>x10, x11, x12, x13, x14, x15, x16, x17, x18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x20, x21, x22, x23, x24, x25, x26, x27, x28</td>
<td>x30, x31, x32, x33, x34, x35, x36, x37, x38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x40, x41, x42, x43, x44, x45, x46, x47, x48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
This table creates AllDiff-constraints for all arrays listed in every row of this decision table. This completes the decision model. To execute this model we may use this Java method:

```java
public static void main(String[] args) {
    String fileName = "file:rules/Decision.xls";
    System.setProperty("OPENRULES_MODE", "Solve");
    Decision decision = new Decision("DefineAndSolveSudoku",fileName);
    decision.execute();
    decision.execute("PrintSolution",decision);
}
```

To print a solution we will use this Excel’s table:
We will receive the following results:

```java
*** Decision DefineAndSolveSudoku ***
Create RuleSolver
JSR-331 Implementation based on Constrainer 5.4 (light)
=== Rule Solver (version 6.2.0) ===
addConstrainedVariables
Decision has been initialized with RuleSolver
=== Initial Problem Variables:
Decision DefineAndSolveSudoku: Create 9x9 Matrix
Decision DefineAndSolveSudoku: Enter Known Problem Data
Decision DefineAndSolveSudoku: Define Main Constraints
=== After Assigning Data:
=== After Posting Constraints:
=== Solve ===
========= Solution =========
1  6  9  8  2  7  3  4  5
7  5  8  1  3  4  9  2  6
4  3  2  9  5  6  1  7  8
8  9  5  7  4  1  6  3  2
6  7  3  2  9  5  8  1  4
2  4  1  6  8  3  7  5  9
3  2  6  5  1  8  4  9  7
9  1  7  4  6  2  5  8  3
5  8  4  3  7  9  2  6  1
============================= 
```

To appreciate the expressiveness of the problem representation supported by Rule Solver™ you may compare it with a pure Java representation - see Sudoku.java in org.jcp.jsr331.samples.

Magic Square Problem
Let’s consider a famous “magic square” problem.

A magic square is a square matrix where the sum of every row, column, and diagonal is equal to the same value. The numbers in the magic square are consecutive and start with 1.

See an example of the Magic Square located in the Passion Façade of the famous Sagrada Familia temple in Barcelona.

This problem can be defined and solved using the following Excel table:

```
Method void main(RuleSolver rs)
    int n = 4;
    VarMatrix matrix =
        rs.variableMatrix("Square",1,n*n,n,n);
    // post AllDiff constraint
    rs.postAllDiff(matrix.flat());
    // post Sum constraints for rows, columns, and diagonals
    int sum = n * (n * n + 1) / 2;
    rs.post(matrix.diagonall(),"=" ,sum);
    rs.post(matrix.diagonall(),"=" ,sum);
    for (int i = 0; i < n; i++) {
        rs.post(matrix.row(i),"=" ,sum);
        rs.post(matrix.column(i),"=" ,sum);
    }
    findSolution(rs);
    System.out.println(matrix);
```

Zebra Problem

This problem is often called "Einstein's Riddle" because it is said to have been invented by Albert Einstein as a boy. Some claim that Einstein said "only 2 percent of the world's population can solve it".

Here are the problem constraints:

1. There are five houses
2. The Englishman lives in the red house
3. The Spaniard owns the dog
4. Coffee is drunk in the green house
5. The Ukrainian drinks tea
6. The green house is immediately to the right of the ivory house
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house
9. Milk is drunk in the middle house
10. The Norwegian lives in the first house
11. The man who smokes Chesterfields lives in the house next to the man with the fox
12. Kools are smoked in the house next to the house where the horse is kept
13. The Lucky Strike smoker drinks orange juice
14. The Japanese smokes Parliaments
15. The Norwegian lives next to the blue house

Where is the Zebra?

Let’s start with a glossary:

<table>
<thead>
<tr>
<th>Glossary glossary</th>
<th>Business Concept</th>
<th>Attribute</th>
<th>Domain</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Colors</strong></td>
<td>Zebra Problem</td>
<td>green</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>green</td>
<td>ivory</td>
<td>ivory</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>blue</td>
<td>red</td>
<td>red</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>yellow</td>
<td>yellow</td>
<td>yellow</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>Norwegian</td>
<td>norwegian</td>
<td>norwegian</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>Ukrainian</td>
<td>ukrainian</td>
<td>ukrainian</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>Japanese</td>
<td>japanese</td>
<td>japanese</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>Englishman</td>
<td>englishman</td>
<td>englishman</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>Spaniard</td>
<td>spaniard</td>
<td>spaniard</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>juice</td>
<td>juice</td>
<td>juice</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>tea</td>
<td>milk</td>
<td>milk</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>water</td>
<td>coffee</td>
<td>coffee</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>snail</td>
<td>dog</td>
<td>dog</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>fox</td>
<td>horse</td>
<td>horse</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>ZEBRA</td>
<td>zebra</td>
<td>zebra</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>Chesterfield</td>
<td>chesterfield</td>
<td>chesterfield</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>Parliament</td>
<td>parliament</td>
<td>parliament</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>Lucky</td>
<td>lucky</td>
<td>lucky</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>OldGolds</td>
<td>oldGolds</td>
<td>oldGolds</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
<tr>
<td>Kools</td>
<td>kools</td>
<td>kools</td>
<td>0-4</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
It defines our decision variables in the domain from 0 to 4 assuming that our houses are numbered as 0, 1, 2, 3, and 4.

We will define AllDiff-constraints for all variables using this decision table:

<table>
<thead>
<tr>
<th>Variables</th>
<th>ActionAllDiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>green, ivory, blue, red, yellow</td>
<td></td>
</tr>
<tr>
<td>Norwegian, Ukrainian, Japanese, Englishman, Spaniard</td>
<td></td>
</tr>
<tr>
<td>juice, tea, milk, water, coffee</td>
<td></td>
</tr>
<tr>
<td>snail, dog, fox, horse, ZEBRA</td>
<td></td>
</tr>
<tr>
<td>Chesterfield, Parliament, Lucky, OldGolds, Kools</td>
<td></td>
</tr>
</tbody>
</table>

We may define unconditional linear constraints using the following decision table:

<table>
<thead>
<tr>
<th>ActionXoperY</th>
<th>Variables</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = red</td>
<td>Englishman</td>
<td>The Englishman lives in the red house</td>
</tr>
<tr>
<td></td>
<td>Spaniard</td>
<td>The Spaniard owns the dog</td>
</tr>
<tr>
<td></td>
<td>coffee</td>
<td>Coffee is drunk in the green house</td>
</tr>
<tr>
<td></td>
<td>Ukrainian</td>
<td>The Ukrainian drinks tea</td>
</tr>
<tr>
<td></td>
<td>OldGolds</td>
<td>The Old Golds smoker owns snails</td>
</tr>
<tr>
<td></td>
<td>Kools</td>
<td>Kools are smoked in the yellow house</td>
</tr>
<tr>
<td></td>
<td>milk</td>
<td>Milk is drunk in the middle house</td>
</tr>
<tr>
<td></td>
<td>Norwegian</td>
<td>The Norwegian lives in the first house</td>
</tr>
<tr>
<td></td>
<td>Lucky</td>
<td>The Lucky Strike smoker drinks orange juice</td>
</tr>
<tr>
<td></td>
<td>Japanese</td>
<td>The Japanese smokes Parliament</td>
</tr>
</tbody>
</table>

We will use formulas with JSR-331 code to define neighboring constraints:
To put everything together we will use the following table “Decision”:

<table>
<thead>
<tr>
<th>Decisions</th>
<th>Execute Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Diff Constraints</td>
<td>AllDiffConstraints</td>
</tr>
<tr>
<td>Zebra Constraints 1</td>
<td>ZebraConstraints1</td>
</tr>
<tr>
<td>Zebra Constraints 2</td>
<td>ZebraConstraints2</td>
</tr>
</tbody>
</table>

We will assume that data comes from Java that uses the following class:

```java
public class ZebraProblem {

    int green, ivory, blue, red, yellow;
    int norwegian, ukrainian, japanese, englishman, spaniard;
    int juice, tea, milk, water, coffee;
    int snail, dog, fox, horse, zebra;
    int chesterfield, parliament, lucky, oldGolds, kools;

    // automatically generated getters, setters, and toString()
}
```
The glossary will be mapped with the actual Java objects using this table:

<table>
<thead>
<tr>
<th>Business Concept</th>
<th>Business Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zebra Problem</td>
<td>decision.get(&quot;problem&quot;)</td>
</tr>
</tbody>
</table>

Now we are ready to execute this Java launcher:

```java
public static void main(String[] args) {
    String fileName = "file:rules/main/Decision.xls";
    System.setProperty("OPENRULES_MODE", "Solve");
    Decision decision = new Decision("FindZebra", fileName);
    ZebraProblem zp = new ZebraProblem();
    decision.put("problem", zp);
    decision.put("trace","On");
    decision.execute();
    out.println("\nDecision: " + zp);
    decision.execute("printSolution",decision);
}
```

To print a solution it will use the following method:

```
Method void printSolution(Decision decision)
```

It will produce the following results:

```txt
*** Decision FindZebra ***
Create RuleSolver
JSR-331 Standard v.1.0.1 (release 5/25/2012)
JSR-331 Implementation based on Constrainer 5.4 (light)
=== Rule Solver (version 6.2.1) ===
addConstrainedVariables
Decision has been initialized with RuleSolver
=== Initial Problem Variables:
Lucky[0..4]
Chesterfield[0..4]
```
fo\[0..4\]
OldGolds[0..4]
Ukrainian[0..4]
horse[0..4]
Parliament[0..4]
yellow[0..4]
te[0..4]
Kools[0..4]
milk[0..4]
juice[0..4]
ivory[0..4]
water[0..4]
Japanese[0..4]
ZEBRA[0..4]
green[0..4]
coffee[0..4]
Norwegian[0..4]
red[0..4]
blue[0..4]
Englishman[0..4]
Spaniard[0..4]

dog[0..4]
Decision FindZebra: All Diff Constraints
Decision FindZebra: Zebra Constraints 1
Decision FindZebra: Zebra Constraints 2

=== After Assigning Data:
...
=== After Posting Constraints:
Lucky[0..4]
Chesterfield[0..4]
fox[0..4]
OldGolds[0..4]
Ukrainian[1..4]
horse[0..4]
Parliament[1..4]
yellow[0..4]
tea[1..4]
Kools[0..4]
milk[2]
juice[0..4]
ivory[2..3]
water[0..4]
Japanese[1..4]
ZEBRA[0..4]
green[3..4]
coffee[3..4]
Norwegian[0]
red[2..4]
blue[1]
Englishman[2..4]
Spaniard[1..4]
snail[0..4]
dog[1..4]

=== Solve ===

... Decision: ZebraProblem
green=4, ivory=3, blue=1, red=2, yellow=0
Norwegian=0, Ukrainian=1, Japanese=4, Englishman=2, Spaniard=3
juice=3, tea=1, milk=2, water=0, coffee=4
snail=2, dog=3, fox=0, horse=1, ZEBRA=4
Chesterfield=1, Parliament=4, Lucky=3, OldGolds=2, Kools=0

House #1: fox yellow Kools water Norwegian
House #2: Chesterfield Ukrainian horse tea blue
House #3: OldGolds milk red Englishman snail
House #4: Lucky juice ivory Spaniard dog
House #5: Parliament Japanese ZEBRA green coffee

The proper project “DecisionZebra” is a part of the standard Rule Solver™ installation.

Solving Scheduling Problems

The current version of the JSR-331 does not yet include a scheduling package. So, OpenRules® developed a simple package javax.constraints.scheduler on the top of the standard JSR-331 interface. It allows a user to define and solve different scheduling and resource allocation problems. Rule Solver™ also provides OpenRules® Excel templates that utilize the package org.jcp.jsr331.scheduler, allowing a user to present scheduling problems directly in Excel.

General Model

Scheduling is the process of placing activities in proper time sequence and allocating the correct resources to activities over time. While there is a great diversity in scheduling problems, there are a lot of constraints and strategies that are common for many problems. The package
javax.constraints.scheduler implements the following general scheduling model:

A scheduling problem can be defined in terms of activities and resources. Activities may have an unknown start, duration and end, and may require (or produce) different resources. Resources may have different types (e.g. recoverable like humans or consumable like money) and their limited capacities may vary over time. The package includes major temporal and capacity constraints specified in accordance with the JSR-331 requirements.

We will describe this package below. Now we will consider the following examples of scheduling problems presented in Java or with Rule Solver™ templates.

Example “Scheduling Construction Jobs”

Let’s assume that we plan to construct a house. The construction requires the following activities with the following fixed durations and precedence (temporal) constraints:
Here arrows represent temporal constraints like “Carpentry starts after the end of Masonry”. The numbers near each activity represent its durations (in days).

**Solution in Java**

First consider a pure Java code for this simple problem.

```java
package org.jcp.jsr331.scheduler.samples;

import javax.constraints.*;
import javax.constraints.scheduler.*;

public final class ScheduleActivities {

    Schedule s = ScheduleFactory.newSchedule("ScheduleActivities", 0, 40);

    public void define() throws Exception {

        // defining jobs
        Activity masonry = s.activity("masonry", 7);
        Activity carpentry = s.activity("carpentry", 3);
        Activity plumbing = s.activity("plumbing", 8);
        Activity ceiling = s.activity("ceiling", 3);
        Activity roofing = s.activity("roofing", 1);
        Activity painting = s.activity("painting", 2);
        Activity windows = s.activity("windows", 1);
        Activity facade = s.activity("facade", 2);
        Activity garden = s.activity("garden", 1);
        Activity movingIn = s.activity("moving in", 1);
    }
}
```
// Posting "startsAfterEnd" constraints
s.post(carpentry,">",masonry);
s.post(roofing,">",carpentry);
s.post(plumbing,">",masonry);
s.post(ceiling,">",masonry);
s.post(windows,">",roofing);
s.post(facade,">",roofing);
s.post(facade,">",plumbing);
s.post(garden,">",roofing);
s.post(garden,">",plumbing);
s.post(painting,">",ceiling);
s.post(movingIn,">",windows);
s.post(movingIn,">",facade);
s.post(movingIn,">",garden);
s.post(movingIn,">",painting);

s.logActivities();
}

public void solve() {
    Solution solution = s.scheduleActivities();
    if (solution == null)
        s.log("No solutions");
    else {
        s.log("SOLUTION:");
        s.logActivities();
    }
}

public static void main(String args[]) throws Exception {
    ScheduleActivities p = new ScheduleActivities();
p.define();
p.solve();
}

The line
    s.post(plumbing,">",masonry);
posts the constraint “Carpentry starts after the end of Masonry”. The method
“solve” uses a predefined method, “scheduleActivities”, that simply defines start
times for all activities while satisfying all posted constraints. When we run this
code, it will produce:

SOLUTION:
masonry [0 -- 7 --> 7)
carpentry[7 -- 3 --> 10)
plumbing [7 -- 8 --> 15)
ceiling  [7 -- 3 --> 10)
roofing  [10 -- 1 --> 11)
painting [10 -- 2 --> 12]  
windows  [11 -- 1 --> 12]  
facade  [15 -- 2 --> 17]  
garden  [15 -- 1 --> 16]  
moving in[17 -- 1 --> 18]

Here the line movingIn[17 -- 1 --> 18] means that the activity “movingIn” will start on day 17, will last one day, and will end on day 18.

Solution in Excel

Now we will see how the same problem can be presented and solved directly in Excel. Rule Solver™ provides s for different scheduling constructs. So, our decision model will be defined by the following table:

<table>
<thead>
<tr>
<th>Decision ScheduleActivities</th>
<th>Execute Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define Schedule</td>
<td>DefineSchedule</td>
</tr>
<tr>
<td>Define Activities</td>
<td>DefineActivities</td>
</tr>
<tr>
<td>Define Precedence Constraints</td>
<td>DefinePrecedenceConstraints</td>
</tr>
</tbody>
</table>

We may define schedule in the following table:

<table>
<thead>
<tr>
<th>DecisionTable DefineSchedule</th>
<th>ActionSchedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin</td>
<td>Horizon</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
</tr>
</tbody>
</table>

Then we will define activities:

<table>
<thead>
<tr>
<th>DecisionTable DefineActivities</th>
<th>ActionAddActivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Duration</td>
</tr>
<tr>
<td>masonry</td>
<td>7</td>
</tr>
<tr>
<td>carpentry</td>
<td>3</td>
</tr>
<tr>
<td>roofing</td>
<td>1</td>
</tr>
<tr>
<td>plumbing</td>
<td>8</td>
</tr>
<tr>
<td>ceiling</td>
<td>3</td>
</tr>
<tr>
<td>windows</td>
<td>1</td>
</tr>
<tr>
<td>façade</td>
<td>2</td>
</tr>
</tbody>
</table>
And finally we will define precedence constraints:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Operator</th>
<th>Activity/Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>carpentry</td>
<td>&gt;</td>
<td>masonry</td>
</tr>
<tr>
<td>roofing</td>
<td>&gt;</td>
<td>carpentry</td>
</tr>
<tr>
<td>plumbing</td>
<td>&gt;</td>
<td>masonry</td>
</tr>
<tr>
<td>ceiling</td>
<td>&gt;</td>
<td>masonry</td>
</tr>
<tr>
<td>windows</td>
<td>&gt;</td>
<td>roofing</td>
</tr>
<tr>
<td>façade</td>
<td>&gt;</td>
<td>plumbing</td>
</tr>
<tr>
<td>façade</td>
<td>&gt;</td>
<td>roofing</td>
</tr>
<tr>
<td>garden</td>
<td>&gt;</td>
<td>roofing</td>
</tr>
<tr>
<td>garden</td>
<td>&gt;</td>
<td>plumbing</td>
</tr>
<tr>
<td>movingIn</td>
<td>&gt;</td>
<td>windows</td>
</tr>
<tr>
<td>movingIn</td>
<td>&gt;</td>
<td>façade</td>
</tr>
<tr>
<td>movingIn</td>
<td>&gt;</td>
<td>garden</td>
</tr>
<tr>
<td>movingIn</td>
<td>&gt;</td>
<td>painting</td>
</tr>
</tbody>
</table>

The model is ready to be executed by this Java launcher:

```java
public static void main(String[] args) {
    String fileName = "file:rules/Decision.xls";
    System.setProperty("OPENRULES_MODE", "Solve");
    Decision decision = new Decision("ScheduleActivities",fileName);
    decision.execute();
}
```

It will produce the following results:

*** Decision ScheduleActivities ***
Create RuleSolver
JSR-331 Implementation based on Constrainer 5.4 (light)
=== Rule Solver (version 6.2.0) ===
addConstrainedVariables
Decision has been initialized with RuleSolver
=== Initial Problem Variables:
Decision ScheduleActivities: Define Schedule
Create RuleScheduler
Decision ScheduleActivities: Define Activities
Decision ScheduleActivities: Define Precedence Constraints
Example “Resource Allocation”

The following problem deals with activities that require a common resource. Let’s consider 5 different orders (activities) that fire batches of bricks in an oven (a resource with a limited capacity). Each order’s size and duration, as well as the oven’s capacity, are described in the following figure:
This is a simple example of a joint scheduling and resource allocation problem where we allow a solver to decide when to perform different activities based on resource availability.

**Solution in Excel**

Now we will see how this problem can be presented and solved directly in Excel. Let’s define our decision model:

<table>
<thead>
<tr>
<th>Decision</th>
<th>Execute Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define Schedule</td>
<td>DefineSchedule</td>
</tr>
<tr>
<td>Define Activities</td>
<td>DefineActivities</td>
</tr>
<tr>
<td>Define Oven as Recoverable Resource</td>
<td>DefineOvenAsResource</td>
</tr>
<tr>
<td>Define Oven Availability</td>
<td>SetOvenCapacities</td>
</tr>
<tr>
<td>Define Resource Requirement Constraints</td>
<td>ResourceRequirementConstraints</td>
</tr>
</tbody>
</table>

This decision table creates a schedule with a makespan 11 days:

<table>
<thead>
<tr>
<th>ActionSchedule</th>
<th>Origin</th>
<th>Horizon</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefineSchedule</td>
<td>0</td>
<td>11</td>
</tr>
</tbody>
</table>

This table defines all activities:

<table>
<thead>
<tr>
<th>ActionAddActivity</th>
<th>Name</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddActivity</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>4</td>
</tr>
</tbody>
</table>
The resource “Oven” can be created using this table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Max Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oven</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

This problem does not have precedence constraints but it has resource requirement constraints that are presented in this table:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Required Resource</th>
<th>Required Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Oven</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>Oven</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>Oven</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>Oven</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>Oven</td>
<td>2</td>
</tr>
</tbody>
</table>

The resource “Oven” has limits to its capacities as defined in the following table:

<table>
<thead>
<tr>
<th>Resource</th>
<th>From</th>
<th>To</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oven</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Oven</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Oven</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Oven</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Oven</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Oven</td>
<td>10</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

The model is ready to be executed by this Java launcher:

```java
public static void main(String[] args) {
    String fileName = "file:rules/Decision.xls";
    System.setProperty("OPENRULES_MODE", "Solve");
    Decision decision = new Decision("DefineOvenSchedule", fileName);
    decision.execute();
}
```
It will produce the following results:

*** Decision DefineOvenSchedule ***
Create RuleSolver
JSR-331 Implementation based on Constrainer 5.4 (light)
=== Rule Solver (version 6.2.0) ===
addConstrainedVariables
Decision has been initialized with RuleSolver
=== Initial Problem Variables: ===
Decision DefineOvenSchedule: Define Schedule
Create RuleScheduler
Decision DefineOvenSchedule: Define Activities
Decision DefineOvenSchedule: Define Oven as Recoverable Resource
Decision DefineOvenSchedule: Define Oven Availability
Decision DefineOvenSchedule: Define Resource Requirement Constraints
=== After Assigning Data: ===
=== After Posting Constraints: ===
=== Solve ===
Solution:
A[5 -- 1 --> 6) requires Oven[2]
B[3 -- 4 --> 7) requires Oven[1]
C[7 -- 4 --> 11) requires Oven[1]
D[0 -- 2 --> 2) requires Oven[1]
E[6 -- 4 --> 10) requires Oven[2]

Here is a visual representation of the results:

![Solution in Java](image)

Solution in Java

Now we solve the same problem in Java:

```java
package org.jcp.jsr331.scheduler.samples;

import javax.constraints.Solution;
import javax.constraints.Solver;
import javax.constraints.scheduler.Activity;
import javax.constraints.scheduler.Resource;
import javax.constraints.scheduler.impl.SchedulingProblem;
```
public final class Oven {

    Schedule s =
        ScheduleFactory.newSchedule("Oven", 0, 40);

    public void define() throws Exception {
        setStart(0);
        setEnd(11);

        Activity A = s.activity("A", 1);
        Activity B = s.activity("B", 4);
        Activity C = s.activity("C", 4);
        Activity D = s.activity("D", 2);
        Activity E = s.activity("E", 4);

        Resource oven = s.resource("oven", 3);

        oven.setCapacityMax(0, 2);
        oven.setCapacityMax(1, 1);
        oven.setCapacityMax(2, 0);
        oven.setCapacityMax(3, 1);
        oven.setCapacityMax(4, 1);
        oven.setCapacityMax(10, 1);

        A.requires(oven, 2);
        B.requires(oven, 1);
        C.requires(oven, 1);
        D.requires(oven, 1);
        E.requires(oven, 2);
    }

    public void solve() {
        Solver solver = s.getSolver();
        solver.setSearchStrategy(s.strategyScheduleActivities());
        solver.addSearchStrategy(s.strategyAssignResources());
        Solution solution = solver.findSolution();
        if (solution == null) {
            s.log("No Solutions");
        } else {
            s.log("Solution:");
            s.logActivities();
            s.logResources();
        }
        solver.logStats();
    }

    public static void main(String args[]) throws Exception {
        Oven p = new Oven();
        p.define();
        p.solve();
    }
}
Here, the line

```java
Resource oven = s.resource("oven", 3);
```

defines a new discrete resource “oven” with a maximum capacity of 3 batches. The following lines set the maximal capacity for the oven for every day when the oven has a capacity less than 3. Note that the solving method along with the strategy “strategyScheduleActivities” also use another strategy “strategyAssignResources”. Putting these two strategies in the solver strategy list, we allow a solver to first try to schedule activities and then to try to assign the required resource capacities to them. When resource assignments fail, a solver will backtrack and will try different placements of activities in time. Here are the produced results:

```plaintext
Solution:
C[7 -- 4 --> 11] requires oven[1]
D[0 -- 2 --> 2] requires oven[1]
```

*** Execution Profile ***
Number of Choice Points: 5
Number of Failures: 2
Execution time: 31 msec

Now the method “logActivities” also shows the required resources with required capacities in brackets. The method “logResources” shows the resource “oven” with its daily capacities assigned to their possible minimums.

The produced statistics shows that there were two failures when the solver had to reconsider previously selected start times for some activities.

**Learn By Examples**

We can extend the previously described basic construction job scheduling example to demonstrate more complex scheduling and resource allocation problems.
Example “Scheduling Construction Jobs with a Worker”

Previously we had pure scheduling constraints between different construction activities. Now let’s assume that additionally all of the activities require a resource (worker) in order to be processed. One worker is required to perform all the activities. Because the worker can only perform one task at a time, we cannot schedule two activities at the same time (as we could in the basic example).

In addition to the decision tables that define activities and precedence constraints in the pure scheduling example, we will need to add a resource “Worker” and the proper requirement constraints. Here is the updated decision:

<table>
<thead>
<tr>
<th>Decision ScheduleActivitiesWithWorker</th>
<th>Execute Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define Schedule</td>
<td>DefineSchedule</td>
</tr>
<tr>
<td>Define Activities</td>
<td>DefineActivities</td>
</tr>
<tr>
<td>Define Precedence Constraints</td>
<td>DefinePrecedenceConstraints</td>
</tr>
<tr>
<td>Define Worker</td>
<td>DefineWorker</td>
</tr>
<tr>
<td>Define Resource Requirement Constraints</td>
<td>ResourceRequirementConstraints</td>
</tr>
</tbody>
</table>

We will add a worker using this table:

<table>
<thead>
<tr>
<th>DecisionTable DefineWorker</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionAddResource</td>
<td></td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>Worker</td>
<td>Recoverable</td>
</tr>
</tbody>
</table>

The next table defines the worker requirement constraints:

<table>
<thead>
<tr>
<th>DecisionTable ResourceRequirementConstraints</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionActReqResource</td>
<td></td>
</tr>
<tr>
<td><strong>Activity</strong></td>
<td><strong>Required Resource</strong></td>
</tr>
<tr>
<td>masonry</td>
<td>Worker</td>
</tr>
<tr>
<td>carpentry</td>
<td>Worker</td>
</tr>
<tr>
<td>roofing</td>
<td>Worker</td>
</tr>
</tbody>
</table>
**Example “Scheduling Construction Jobs with a Limited Budget”**

Now we will add an additional requirement to the above problem. Along with worker constraints, we have to consider budget constraints. Each activity requires the payment of $1,000 per day. Let’s assume that a bank agreed to finance the house constructions for the total amount of $29,000. However, the sum is available in two installations, $13,000 is available at the start of the
project, and $16,000 is available 15 days afterwards. How could we still construct the house under these constraints?

We need to extend our decision model by adding a resource “Budget” and the proper requirement constraints. Here is the updated decision:

<table>
<thead>
<tr>
<th>Decision ScheduleActivitiesWithWorkerBudget</th>
<th>Execute Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define Schedule</td>
<td>DefineSchedule</td>
</tr>
<tr>
<td>Define Activities</td>
<td>DefineActivities</td>
</tr>
<tr>
<td>Define Precedence Constraints</td>
<td>DefinePrecedenceConstraints</td>
</tr>
<tr>
<td>Define Worker &amp; Budget</td>
<td>DefineResources</td>
</tr>
<tr>
<td>Define Budget Limitations</td>
<td>SetBudgetCapacities</td>
</tr>
<tr>
<td>Define Resource Requirement Constraints</td>
<td>ResourceRequirementConstraints</td>
</tr>
</tbody>
</table>

Now we will add both Worker and Budget resources in the following table:

<table>
<thead>
<tr>
<th>DecisionTable DefineResources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Max Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worker</td>
<td>Recoverable</td>
<td>1</td>
</tr>
<tr>
<td>Budget</td>
<td>Consumable</td>
<td>30000</td>
</tr>
</tbody>
</table>

The updated resource requirement table will look as follows:

<table>
<thead>
<tr>
<th>DecisionTable ResourceRequirementConstraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionActReqResource</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Activity</th>
<th>Required Resource</th>
<th>Required Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>masonry</td>
<td>Worker</td>
<td>1</td>
</tr>
<tr>
<td>carpentry</td>
<td>Worker</td>
<td>1</td>
</tr>
<tr>
<td>roofing</td>
<td>Worker</td>
<td>1</td>
</tr>
<tr>
<td>plumbing</td>
<td>Worker</td>
<td>1</td>
</tr>
<tr>
<td>ceiling</td>
<td>Worker</td>
<td>1</td>
</tr>
<tr>
<td>windows</td>
<td>Worker</td>
<td>1</td>
</tr>
<tr>
<td>façade</td>
<td>Worker</td>
<td>1</td>
</tr>
</tbody>
</table>
And finally the fact that during the first 15 days only $15,000 are available can be expressed using the following table:

<table>
<thead>
<tr>
<th>ActionResourceCap</th>
<th>From</th>
<th>To</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budget</td>
<td>0</td>
<td>15</td>
<td>15000</td>
</tr>
</tbody>
</table>

Now the same Java launcher will produce the following results:

```plaintext
*** Decision ScheduleActivitiesWithWorkerBudget ***
Create RuleSolver
JSR-331 Implementation based on Constrainer 5.4 (light)
=== Rule Solver (version 6.2.0) ===
addConstrainedVariables
Decision has been initialized with RuleSolver
=== Initial Problem Variables:
Decision ScheduleActivitiesWithWorkerBudget: Define Schedule
Create RuleScheduler
Decision ScheduleActivitiesWithWorkerBudget: Define Activities
Decision ScheduleActivitiesWithWorkerBudget: Define Precedence Constraints
Decision ScheduleActivitiesWithWorkerBudget: Define Worker & Budget
Decision ScheduleActivitiesWithWorkerBudget: Define Budget Limitations
Decision ScheduleActivitiesWithWorkerBudget: Define Resource Requirement Constraints
=== After Assigning Data:
=== After Posting Constraints:
=== Solve ===
Solution:
masonry[0 -- 7 --> 7) requires Worker[1] requires Budget[1000]
carpentry[7 -- 3 --> 10) requires Worker[1] requires Budget[1000]
roofing[10 -- 1 --> 11) requires Worker[1] requires Budget[1000]
plumbing[11 -- 8 --> 19) requires Worker[1] requires Budget[1000]
ceiling[19 -- 3 --> 22) requires Worker[1] requires Budget[1000]
windows[22 -- 1 --> 23) requires Worker[1] requires Budget[1000]
```
façade[23 -- 2 --> 25) requires Worker[1]  requires Budget[1000]
garden[25 -- 1 --> 26) requires Worker[1]  requires Budget[1000]
painting[26 -- 2 --> 28) requires Worker[1]  requires Budget[1000]
movingIn[28 -- 1 --> 29) requires Worker[1]  requires Budget[1000]

Example “Scheduling Construction Jobs with Alternative Resources”

Now let’s consider a construction job scheduling example with alternative resources that are required to execute those jobs. Let’s assume that we have 3 workers Joe, Jim, and Jack, with different skills. Each job requires only one of these workers depending on their skills:

- masonry    requires Joe or Jack
- carpentry  requires Joe or Jim
- plumbing   requires Jack
- ceiling    requires Joe or Jim
- roofing     requires Jack or Jim
- windows    requires Joe or Jim
- façade     requires Joe or Jack
- garden     requires Joe or Jack or Jim
- movingIn   requires Joe or Jim.

We will extend the previously described basic construction job scheduling example. We need to add 3 disjunctive resources, Joe, Jim, and Jack. The fact that an activity may require more than one resource is interpreted as a requirement for alternative resources. We will add sub-decision “Define Workers” and “Define Resource Requirement Constraints” to the table “Decision”:

<table>
<thead>
<tr>
<th>Decisions</th>
<th>Execute Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define Schedule</td>
<td>DefineSchedule</td>
</tr>
<tr>
<td>Define Activities</td>
<td>DefineActivities</td>
</tr>
<tr>
<td>Define Precedence Constraints</td>
<td>DefinePrecedenceConstraints</td>
</tr>
<tr>
<td>Define Workers</td>
<td>DefineWorkers</td>
</tr>
<tr>
<td>Define Resource Requirement Constraints</td>
<td>ResourceRequirementConstraints</td>
</tr>
</tbody>
</table>
The following table defines alternative resources:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Max Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>disjunctive</td>
<td>1</td>
</tr>
<tr>
<td>Jack</td>
<td>disjunctive</td>
<td>1</td>
</tr>
<tr>
<td>Jim</td>
<td>disjunctive</td>
<td>1</td>
</tr>
</tbody>
</table>

The next table creates resource requirement constraints listing alternatives divided by the OR-sign “|”:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Required Resource</th>
<th>Required Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>masonry</td>
<td>Joe</td>
<td>Jack</td>
</tr>
<tr>
<td>carpentry</td>
<td>Joe</td>
<td>Jim</td>
</tr>
<tr>
<td>roofing</td>
<td>Joe</td>
<td>Jim</td>
</tr>
<tr>
<td>plumbing</td>
<td>Jack</td>
<td></td>
</tr>
<tr>
<td>ceiling</td>
<td>Joe</td>
<td>Jim</td>
</tr>
<tr>
<td>windows</td>
<td>Joe</td>
<td>Jim</td>
</tr>
<tr>
<td>façade</td>
<td>Joe</td>
<td>Jack</td>
</tr>
<tr>
<td>garden</td>
<td>Joe</td>
<td>Jim</td>
</tr>
<tr>
<td>painting</td>
<td>Jack</td>
<td>Jim</td>
</tr>
<tr>
<td>movingIn</td>
<td>Joe</td>
<td>Jim</td>
</tr>
</tbody>
</table>

That's it. Now the same Java launcher will produce the following results:

```
*** Decision ScheduleActivitiesWithAlternativeResources ***
Create RuleSolver
JSR-331 Implementation based on Constrainer 5.4 (light)
*** Rule Solver (version 6.2.0) ***
addConstrainedVariables
Decision has been initialized with RuleSolver
=== Initial Problem Variables:
Decision ScheduleActivitiesWithAlternativeResources: Define Schedule
Create RuleScheduler
Decision ScheduleActivitiesWithAlternativeResources: Define Activities
Decision ScheduleActivitiesWithAlternativeResources: Define Precedence
Constraints
Decision ScheduleActivitiesWithAlternativeResources: Define Workers
Decision ScheduleActivitiesWithAlternativeResources: Define Resource
Requirement Constraints
=== After Assigning Data:
```
### After Posting Constraints:

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Requires</th>
</tr>
</thead>
<tbody>
<tr>
<td>masonry[0 -- 7 --&gt; 7]</td>
<td>Jack[1]</td>
</tr>
<tr>
<td>painting[0 -- 2 --&gt; 2]</td>
<td>Jim[1]</td>
</tr>
<tr>
<td>movingIn[17 -- 1 --&gt; 18]</td>
<td>Jim[1]</td>
</tr>
</tbody>
</table>

## INSTALLATION

Rule Solver™ can be installed as an component of the complete version of OpenRules® and available in the workspace “openrules.solver”. You will download and unzip this folder to your hard drive. It is self-sufficient and can be used with Windows Explorer or Eclipse IDE.

### Structure

Unzipped “openrules.solver” includes the following projects:

**Project “com.openrules.solver“**

This project contains an implementation of Rule Solver™:

- Folder “src” with source files:
  - com.openrules.solver – source code for Rule Solver™
  - com.openrules.solver.samples - Rule Solver™ examples including “MissManners” that deals with set constrained variables
  - org.jcp.jsr331.samples - sources with JSR-331 examples
  - org.jcp.jsr331.scheduler.samples - sources with scheduling examples

- Folder “lib” with supporting jar-files:
  - jsr331.jar – JSR-331 jar files with 3 open source implementations
- o apache/*.jar – Apache Commons jars
  - o choco/*.jar – jars for Choco's implementation of the JSR-331
  - o jacop/*.jar – jars for JaCoP’s implementation of the JSR-331
  - o constrainer/*.jar – jars for Constrainer’s implementation of the JSR-331
  - o scheduler.jar – Scheduler’s jar file that also includes all sources

- Folder “rules” with Excel-files for all Rule Solver™ samples:
  - o common/Templates.xls – basic OpenRules® templates for defining and solving CSPs
  - o common/ScheduleTemplates.xls – OpenRules® templates for scheduling CSPs
  - o common/RulesSolver.xls – common Excel files that defines Rule Solver™ environment (called from all other Excel files)
  - o <Name>.xls – various Rule Solver™ examples (without decisions)

Project “openrules.config“

This project contains standard OpenRules® jars (in the folder “lib”) and decision templates. Additionally to the standard templates, Rule Solver™ comes with one additional template “DecisionTableSolveTemplates.xls”.

Decision Projects

There are many sample projects such as DecisionHelloCP, DecisionLoanCP, DecisionScheduleActivities, and others.

Licenses

Rule Solver™ is available under the terms of the most popular Open Source license known as "GNU General Public License" (GPLv2). The included software is provided under the terms of open source licenses included in the folders for the proper solvers.
Using a Standalone Version

You may use Rule Solver™ directly from your file system. All projects such as DecisionLoanXP contain a batch file “run.bat”.

The folder com.openrules.solver contains batch files that can be used to run different examples. For example, "runZebra.bat" will execute the example Zebra.xls. To run any example, you may double-click on the proper run<Name>.bat file. For example, runOven.bat will execute the problem defined in the file “rules/Oven.xls”.

If you work with UNIX/LINUX you need to replace *.bat files with similar *.sh files.

To switch between CP solvers you need to modify the file “run.bat”. For example, to switch from "constrainer" to "choco" put "rem " in front of "set SOLVER=./lib/constrainer/..." and remove "rem " in front of "set SOLVER=./lib/choco/...".

Working under Eclipse IDE

To use the Rule Solver™ with Eclipse IDE, simply import the project com.openrules.solver and different decision projects such as DecisionLoanXP into your Eclipse workspace. You may run Java samples directly from Eclipse by selecting their sources with a right-click and then "Run as Java Application".

To switch between underlying solvers, just select the Project Properties, and simply change Libraries inside Java Build Path.

TECHNICAL SUPPORT

Direct all your technical questions to support@openrules.com or to these Google discussion groups: OpenRules Forum and JSR-331 Forum.