



RULE SOLVERTM

Constraint Programming

with

OpenRules[®]

USER MANUAL

OpenRules, Inc.

www.openrules.com

March-2012

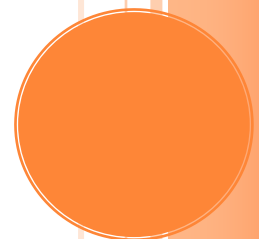


Table of Contents

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	26
Solving the Problem	26
Using Templates	26
Using JSR-331	27
Implementation Restrictions and Future Improvements	28
Rule Solver as a Business-Oriented Constraint Solver	29
Constraint Satisfaction Problem (CSP)	29
Formal Definition	29
Major CP Concepts	30
Introductory Example “SEND+MORE=MONEY”	31
Excel-based Decision Model	31
Pure Java Solution	35
Solving Arithmetic Problems	36
Sudoku Problem	39
Magic Square Problem	43
Zebra Problem	43
Solving Scheduling Problems	49
General Model	49
Example “Scheduling Construction Jobs”	50
Solution in Java	51
Solution in Excel	53

Example “Resource Allocation”	55
Solution in Excel	56
Solution in Java	58
Learn By Examples	60
Example “Scheduling Construction Jobs with a Worker”	61
Example “Scheduling Construction Jobs with a Limited Budget”	62
Example “Scheduling Construction Jobs with Alternative Resources”	65
Installation	67
Structure	67
Project “com.openrules.solver”	67
Project “openrules.config”	68
This project contains standard OpenRules® jars (in the folder “lib”) and decision templates.	68
Decision Projects	68
There are many sample projects such as DecisionHelloCP, DecsionLoanCP, DecisionScheduleActivities, and others described above.	68
Licenses	68
Using a Standalone Version	69
Working under Eclipse IDE	69
Technical Support	69

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-331¹ 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 AS INFERENTIAL RULE ENGINE

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:

Decision DetermineCustomerGreeting	
Decisions	Execute Rules
Define Age Group	:= DefineAgeGroup()
Define Greeting Word	:= DefineGreeting()
Define Salutation Word	:= DefineSalutation()

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

DecisionTable DefineAgeGroup					
Condition		Condition		Conclusion	
Age		Age		Age Group	
>=	0	<=	5	Is	Little
>	5	<=	20	Is	Young
>	20			Is	Adult

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

DecisionTable DefineGreeting					
Condition		Condition		Conclusion	
Current Hour		Current Hour		Greeting	
>=	0	<=	11	Is	Good Morning
>	11	<=	17	Is	Good Afternoon
>	17	<=	22	Is	Good Evening
>	22	<=	24	Is	Good Night

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

DecisionTable DefineSalutation							
Condition		Condition		Condition		Conclusion	
Gender		Marital Status		Age Group		Salutation	
				Is	Little	Is	Little
Is	Female	Is	Married	Is Not	Little	Is	Mrs.
Is	Female	Is	Single			Is	Ms.
Is	Male					Is	Mr.

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

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

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

DecisionObject decisionObjects	
Business Concept	Business Object
Customer	:= decision.get("customer")
Response	:= decision.get("response")

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:

```

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:

```

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

```
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:

Decision DetermineCustomerGreeting	
Decisions	Execute Rules
Define Greeting Word	:= DefineGreeting()
Define Salutation Word	:= DefineSalutation()
Define Age Group	:= DefineAgeGroup()

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:

```

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() + "!");
    }
}

```

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

```

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:

```

*** 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:

```

=== 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:

Decision DetermineLoanPreQualificationResults	
Decisions	Execute Rules
Calculate Internal Variables	:= CalculateInternalVariables()
Validate Income	:= DetermineIncomeValidationResult()
Debt Research	:= DetermineDebtResearchResult()
Summarize	:= DetermineLoanQualificationResult()

The first sub-decision “Calculate Internal Variables” will be implemented using the following decision table:

DecisionTable CalculateInternalVariables			
Conclusion		Conclusion	
Total Debt		Total Income	
Is	::= getInt("Monthly Debt") * getInt("Loan Term")	Is	::= getInt("Monthly Income") * getInt("Loan Term")

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

DecisionTable DetermineIncomeValidationResult			
Condition		Conclusion	
Total Income		Income Validation Result	
Is More Than	::= getInt("Total Debt") * 2	Is	SUFFICIENT
<=	::= getInt("Total Debt") * 2	Is	UNSUFFICIENT

The third sub-decision “Debt Research” will be implemented using a more complex decision table:

DecisionTable DetermineDebtResearchResult													
Condition		Condition		Condition		Condition		Condition		Condition		Conclusion	
Mortgage Holder		Outside Credit Score		Outside Credit Score		Loan Holder		Credit Card Balance		Education Loan Balance		Internal Credit Rating	
Internal Analyst Opinion		Debt Research Result											
Is	Yes											Is	High
Is	No	>	100	<=	550							Is	High
Is	No	>	550	<=	900	Is	Yes	<=	0			Is	Mid
Is	No	>	550	<=	900	Is	Yes	>	0	>	0	Is	High
Is	No	>	550	<=	900	Is	Yes	>	0	<=	0	Is One Of	A, B, C
Is	No	>	550	<=	900	Is	Yes	>	0	<=	0	Is One Of	D, F
Is	No	>	550	<=	900	Is	No	>	0			Is	Low
Is	No	>	550	<=	900	Is	No	<=	0	<=	0	Is	Low
Is	No	>	550	<=	900	Is	No	<=	0	>	0	Is One Of	D, F
Is	No	>	550	<=	900	Is	No	<=	0	>	0	Is One Of	A, B, C
Is	No											Is	High
Is	No											Is	Mid
Is	No											Is	Low

And the fourth decision “Summarize” will be implemented using the following decision table:

DecisionTable DetermineLoanQualificationResult					
Condition		Condition		Conclusion	
Income Validation Result		Debt Research Result		Loan Qualification Result	
Is	UNSUFFICIENT			Is	NOT QUALIFIED
Is	SUFFICIENT	Is	Low	Is	NOT QUALIFIED
Is	SUFFICIENT	Is One Of	Mid, High	Is	QUALIFIED

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

Glossary glossary			
Decision Variable	Object	Attribute	Domain
Monthly Income	Customer	monthlyIncome	0-5000000
Mortgage Holder		mortgageHolder	Yes,No
Outside Credit Score		outsideCreditScore	0-999
Loan Holder		loanHolder	Yes,No
Credit Card Balance		creditCardBalance	-1000000 - 100000000
Education Loan Balance		educationLoanBalance	-1000000 - 100000000
Internal Credit Rating		internalCreditRating	A,B,C,D,F
Internal Analyst Opinion		internalAnalystOpinion	High,Mid,Low
Income Validation Result	Request	incomeValidationResult	SUFFICIENT,UNSUFFICIENT,?
Debt Research Result		debtResearchResult	High,Mid,Low,?
Loan Qualification Result		loanQualificationResult	QUALIFIED, NOT QUALIFIED, ?
Total Income	Internal	totalIncome	0-500000
Total Debt		totalDebt	0-500000

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”:

Datatype Customer	
String	fullName
String	SSN
int	monthlyIncome
int	monthlyDebt
String	mortgageHolder
int	outsideCreditScore
String	loanHolder
int	creditCardBalance
int	educationLoanBalance
String	internalCreditRating
String	internalAnalystOpinion

The test-customers are defined in the following table:

Glossary glossary			
Decision Variable	Object	Attribute	Domain
Monthly Income	Customer	monthlyIncome	0-5000000
Monthly Debt		monthlyDebt	0-5000000
Mortgage Holder		mortgageHolder	Yes,No
Outside Credit Score		outsideCreditScore	0-999
Loan Holder		loanHolder	Yes,No
Credit Card Balance		creditCardBalance	-1000000 - 100000000
Education Loan Balance		educationLoanBalance	-1000000 - 100000000
Internal Credit Rating		internalCreditRating	A,B,C,D,F
Internal Analyst Opinion		internalAnalystOpinion	High,Mid,Low
Loan Term		Request	term
Income Validation Result	incomeValidationResult		SUFFICIENT,UNSUFFICIENT,?
Debt Research Result	debtResearchResult		High,Mid,Low,?
Loan Qualification Result	loanQualificationResult		QUALIFIED, NOT QUALIFIED, ?
Total Income	Internal	totalIncome	0-500000
Total Debt		totalDebt	0-500000

The following table defines a datatype “LoanRequest”:

Datatype LoanRequest	
String	customer
int	amount
String	purpose
int	term
String	incomeValidationResult
String	debtResearchResult
String	loanQualificationResult

The test-requests are defined in the following table:

Data LoanRequest loanRequests						
customer	amount	purpose	term	incomeValidationResult	debtResearchResult	loanQualificationResult
Customer	Loan Amount	Loan Purpose	Loan Term	Income Validation Result	Debt Research Result	Loan Qualification Result
Peter N. Johnson	30000	Home Improvement	72	?	?	?
Mary K. Brown	15000	Education	36	?	?	?

The following table defines a datatype “InternalVariables”:

Datatype InternalVariables	
int	totalIncome
int	totalDebt

And these variables are created in the following table:

Variable InternalVariables internal	
Total Income	Total Debt
0	0

Now, when the data is specified, 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”:

DecisionObject decisionObjects	
Business Concept	Business Object
Customer	:= customers[0]
Request	:= loanRequests[0]
Internal	:= internal

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 “DecisionLoanCP” in the workspace “openrules.solver”. The main Java class Main.java contains one method “main” that creates and executes our Excel-based decision:

```
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

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

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

```
*** 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
Conclusion: Income Validation Result Is SUFFICIENT
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:

```
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

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

enforces the use of Rule Solver™ instead of the default 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[-1000000..100000000]
Monthly Income[0..5000000]
Credit Card Balance[-1000000..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 $X = \{ X_1, \dots, X_n \}$
- Each business object X_i contains decision variables $V_i = \{ V_1, \dots, V_m \}$ with possible values $D_j = \{ v_{j1}, \dots, v_{jk} \}$ for each variable V_j

- There is a set of rules $R = \{ R_1, \dots, R_r \}$, 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:

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

Then Rule Solver™ adds constrained variables using the following method:

```
solverVar.solver.addConstrainedVariables(getGlossary())
```

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

Then 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 report about conflicts. In many cases constraint posting is sufficient to instantiate all constraint variables (select single values from their domain).

Solving the Problem

However, 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).

Using Templates

Rule Solver™ never generates Java or any other 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 relies on the existing OpenRules® templating 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” rules table. This rule 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 `c1` and `c2` above, and then “AND”s all previously defined conditions similarly to `c1.and(c2)`. 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 `conditionConstraints` 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.

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.
- 3) The next release will also allow a user to specify an optimization objective. It will allow Rule Solver™ to find an optimal decision instead of forcing a user to specify enormous amount of rules to compare all possible decisions.

RULE SOLVER AS A BUSINESS-ORIENTED CONSTRAINT SOLVER

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).

Formal Definition

Formally a constraint satisfaction problem is defined by

- a set of **variables** V_1, V_2, \dots, V_n , and
- a set of **constraints** C_1, C_2, \dots, 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 are at the beginning of the words SEND and MORE. So, first we will define a glossary:

Glossary glossary				
Fact Name	Business Concept	Attribute	Domain	Unknown
S	Puzzle	s	1-9	TRUE
E		e	0-9	TRUE
N		n	0-9	TRUE
D		d	0-9	TRUE
M		m	1-9	TRUE
O		o	0-9	TRUE
R		r	0-9	TRUE
Y		y	0-9	TRUE
SEND		send	1000-9999	TRUE

MORE		more	1000-9999	TRUE
MONEY		money	10000-99999	TRUE

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:

Datatype Puzzle	
int	s
int	e
int	n
int	d
int	m
int	o
int	r
int	y
int	send
int	more
int	money

Variable Puzzle puzzle										
s	e	n	d	m	o	r	y	send	more	money
S	E	N	D	M	O	R	Y	SEND	MORE	MONEY
0	0	0	0	0	0	0	0	0	0	0

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:

DecisionObject decisionObjects	
Business Concept	Business Object
Puzzle	:= puzzle

Now we need to define our intermediate variables SEND, MORE, and MONEY as scalar products using the following table:

DecisionTable AddScalarProducts		
ActionScalProd		
Scalar Product Name	Coefficients	Variables
SEND	1000,100,10, 1	S,E,N,D
MORE	1000,100,10,1	M,O,R,E
MONEY	10000,1000,100,10,1	M,O,N,E,Y

The first row of this decision table defines SEND as a scalar product $S*1000+E*100+N*10+1*D$. Similarly the second and third rows define variables MORE and MONEY using the proper scalar products.

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

DecisionTable MainConstraint				
ActionXoperYcompareZ				
X <oper> Y <compare> Z				
SEND	+	MORE	=	MONEY

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

DecisionTable AllDifferentConstraint
ActionAllDiff
Variables
S, E, N, D, M, O, R, Y

And finally we need a table of the type “Decision” that puts all constraints together:

Decision SendMoreMoney	
Decisions	Execute Rules
Add Intermediate Variables	:= AddScalarProducts()
Main Constraint	:= MainConstraint()
All Different	:= AllDifferentConstraint()

This completes our decision model. To print the result directly from Excel, we may add one more method “PrintSolution.”

```

Method void PrintSolution()
System.out.println(
"\n=====" +
"\n " + getInt("SEND") +
"\n +" + getInt("MORE") +
"\n=====" +
"\n " + getInt("MONEY") +
"\n=====");

```

To execute this model with Rule Solver™ we will use a Java class Main.java with the following main-method:

```

public static void main(String[] args) {

    String fileName = "file:rules/Decision.xls";
    System.setProperty("OPNRULES_MODE", "Solve");
    Decision decision = new Decision("SendMoreMoney", fileName);
    decision.put("trace", "On");
    decision.execute();
    decision.execute("PrintSolution");

}

```

Here are the execution results:

```

*** 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]
MONEY[10000..99999]

```

```

S[1..9]
MORE[1000..9999]
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:

```

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:

```
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:

```
public static void main(String[] args) {

    String fileName = "file:rules/main/Decision.xls";
    System.setProperty("OPNRULES_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:

Glossary glossary				
Decision Variable	Business Concept	Attribute	Domain	Unknown
X	XYZ	x	0-10	TRUE
Y		y	0-10	TRUE
Z		z	0-10	TRUE

We may map the business concept “XYZ” to the actual object of the type XYZ using this DecisionObject table:

DecisionObject decisionObjects	
Business Concept	Business Object
XYZ	:= decision.get("xyz")

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

Decision FindXYZ	
Decisions	Execute Rules
Define Var Z	:= XplusYeqZ()
Binary Constraints	:= BinaryConstraints()

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

DecisionTable BinaryConstraints		
ActionXoperY		
X <oper> Y		
X	<	Y
Z	>	5

DecisionTable XplusYeqZ				
ActionXoperYcompareZ				
X <oper> Y <compare> Z				
X	+	Y	=	Z

Hopefully, these tables are self-explanatory. All these tables extend decision templates predefined in Rule Solver™ file “DecisionTableSolveTemplates.xls” included in the standard installation. After the execution of the above main-method we will receive these results:

```

*** 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

```

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):

Decision DefineAndSolveSudoku	
Decisions	Execute Rules
Create 9x9 Matrix	:= solver().variableMatrix("x",1,9,9,9)
Enter Known Problem Data	:= EnterSudokuData()
Define Main Constraints	:= DefineSudokuConstraints()

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 “x_{ij}”.

We will enter Sudoku data using this Excel method:

```
Method boolean EnterSudokuData()

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:

Data MatrixInt problem1								
Data								
0	6	9	0	0	7	0	0	5
0	5	0	0	0	4	0	2	0
4	0	0	0	5	0	1	0	0
8	0	5	0	0	0	6	0	0
6	7	0	2	9	5	0	1	4
0	0	1	0	0	0	7	0	9
0	0	6	0	1	0	0	0	7
0	1	0	4	0	0	0	8	0
5	0	0	3	0	0	2	6	0

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

DecisionTable DefineSudokuConstraints	
ActionAllDiff	
Variables	
x00,x01,x02,x03,x04,x05,x06,x07,x08	Row Constraints
x10,x11,x12,x13,x14,x15,x16,x17,x18	
x20,x21,x22,x23,x24,x25,x26,x27,x28	
x30,x31,x32,x33,x34,x35,x36,x37,x38	
x40,x41,x42,x43,x44,x45,x46,x47,x48	
x50,x51,x52,x53,x54,x55,x56,x57,x58	
x60,x61,x62,x63,x64,x65,x66,x67,x68	
x70,x71,x72,x73,x74,x75,x76,x77,x78	
x80,x81,x82,x83,x84,x85,x86,x87,x88	
x00,x10,x20,x30,x40,x50,x60,x70,x80	Column Constraints
x01,x11,x21,x31,x41,x51,x61,x71,x81	
x02,x12,x22,x32,x42,x52,x62,x72,x82	
x03,x13,x23,x33,x43,x53,x63,x73,x83	
x04,x14,x24,x34,x44,x54,x64,x74,x84	
x05,x15,x25,x35,x45,x55,x65,x75,x85	
x06,x16,x26,x36,x46,x56,x66,x76,x86	
x07,x17,x27,x37,x47,x57,x67,x77,x87	
x08,x18,x28,x38,x48,x58,x68,x78,x88	
x00,x01,x02,x10,x11,x12,x20,x21,x22	Block Constraints
x03,x04,x05,x13,x14,x15,x23,x24,x25	
x06,x07,x08,x16,x17,x18,x26,x27,x28	
x30,x31,x32,x40,x41,x42,x50,x51,x52	
x33,x34,x35,x43,x44,x45,x53,x54,x55	
x36,x37,x38,x46,x47,x48,x56,x57,x58	
x60,x61,x62,x70,x71,x72,x80,x81,x82	
x63,x64,x65,x73,x74,x75,x83,x84,x85	
x66,x67,x68,x76,x77,x78,x86,x87,x88	

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:

```

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");

}

```

To print a solution we will use this Excel's table:

```

Method void PrintSolution()
System.out.println("===== Solution =====");
System.out.println(solver().getVarMatrix("x"));
System.out.println("=====");

```

We will receive the following results:

```

*** 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 AllDif constraint
rs.postAllDiff(matrix.flat());
// post Sum constraints for rows, columns,
and diagonals
int sum = n * (n * n + 1) / 2;
rs.post(matrix.diagonal1(), "=", sum);
rs.post(matrix.diagonal2(), "=", 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:

Glossary glossary					
	Decision Variables	Business Concept	Attribute	Domain	Unknown
Colors	green	Zebra Problem	green	0-4	TRUE
	ivory		ivory	0-4	TRUE
	blue		blue	0-4	TRUE
	red		red	0-4	TRUE
	yellow		yellow	0-4	TRUE
People	Norwegian		norwegian	0-4	TRUE
	Ukrainian		ukrainian	0-4	TRUE
	Japanese		japanese	0-4	TRUE
	Englishman		englishman	0-4	TRUE
	Spaniard		spaniard	0-4	TRUE
Drinks	juice		juice	0-4	TRUE
	tea		tea	0-4	TRUE
	milk		milk	0-4	TRUE
	water		water	0-4	TRUE
	coffee		coffee	0-4	TRUE
Pets	snail		snail	0-4	TRUE
	dog		dog	0-4	TRUE
	fox		fox	0-4	TRUE
	horse		horse	0-4	TRUE
	ZEBRA		zebra	0-4	TRUE

Cigarettes	Chesterfield	chesterfield	0-4	TRUE
	Parliament	parliament	0-4	TRUE
	Lucky	lucky	0-4	TRUE
	OldGolds	oldGolds	0-4	TRUE
	Kools	kools	0-4	TRUE

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:

DecisionTable AllDiffConstraints	
ActionAllDiff	
Variables	
green,ivory,blue,red,yellow	
Norwegian,Ukrainian,Japanese,Englishman,Spaniard	
juice,tea,milk,water,coffee	
snail,dog,fox,horse,ZEBRA	
Chesterfield,Parliament,Lucky,OldGolds,Kools	

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

DecisionTable ZebraConstraints1		
ActionXoperY		
X <oper> Y		
Englishman	=	red
Spaniard	=	dog
coffee	=	green
Ukrainian	=	tea
OldGolds	=	snail
Kools	=	yellow
milk	=	2
Norwegian	=	0

The Englishman lives in the red house

The Spaniard owns the dog

Coffee is drunk in the green house

The Ukrainian drinks tea

The Old Golds smoker owns snails

Kools are smoked in the yellow house

Milk is drunk in the middle house

The Norwegian lives in the first house

Lucky	=	juice	The Lucky Strike smoker drinks orange juice
Japanese	=	Parliament	The Japanese smokes Parliament

We will use formulas with JSR-331 code to define neighboring constraints:

DecisionTable ZebraConstraints2	
ActionConstraint	
Constraint	
<pre>{ Var green = getVar("green"); Var ivory = getVar("ivory"); solver().linear(green,"=",ivory.plus(1)); }</pre>	The green house is immediately to the right of the ivory house
<pre>{ Var Chesterfield = getVar("Chesterfield"); Var fox = getVar("fox"); solver().linear(Chesterfield,"=",fox.plus(1)).or(solver().linear(Chesterfield,"=",fox.minus(1))); }</pre>	The man who smokes Chesterfields lives in the house next to the man with the fox
<pre>{ Var horse = getVar("horse"); Var Kools = getVar("Kools"); solver().linear(Kools,"=",horse.plus(1)).or(solver().linear(Kools,"=",horse.minus(1))); }</pre>	Kools are smoked in the house next to the house where the horse is kept.
<pre>{ Var Norwegian = getVar("Norwegian"); Var blue = getVar("blue"); solver().linear(Norwegian,"=",blue.plus(1)).or(solver().linear(Norwegian,"=",blue.minus(1))); }</pre>	The Norwegian lives next to the blue house

To put everything together we will use the following table “Decision”:

Decision FindZebra	
Decisions	Execute Rules
All Diff Constraints	:= AllDiffConstraints()
Zebra Constraints 1	:= ZebraConstraints1()
Zebra Constraints 1	:= ZebraConstraints1()

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

```
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:

DecisionObject decisionObjects	
Business Concept	Business Object
Zebra Problem	:= decision.get("problem")

Now we are ready to execute this Java launcher:

```
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.getEngine().run("printSolution");

}
```

To print a solution it will use the following method:

```
Method void printSolution()
for(int house = 0; house < 5; house++) {
    System.out.print("\nHouse
#" + (house+1) + ":");
    Var[] vars =
solver().getVarsWithValue(house);
    for(int i=0; i<vars.length; i++)
        System.out.print(" " + vars[i].getName());
}
```

It will produce the following results:

```

*** Decision FindZebra ***
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:
Lucky[0..4]
Chesterfield[0..4]
fox[0..4]
OldGolds[0..4]
Ukrainian[0..4]
horse[0..4]
Parliament[0..4]
yellow[0..4]
tea[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]
snail[0..4]
dog[0..4]
Decision FindZebra: All Diff Constraints
Decision FindZebra: Zebra Constraints 1
Decision FindZebra: Zebra Constraints 1
=== 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[0..4]
water[0..4]

```

```

Japanese[1..4]
ZEBRA[0..4]
green[0..4]
coffee[0..4]
Norwegian[0]
red[1..4]
blue[0..4]
Englishman[1..4]
Spaniard[1..4]
snail[0..4]
dog[1..4]
=== Solve ===

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

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

```

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

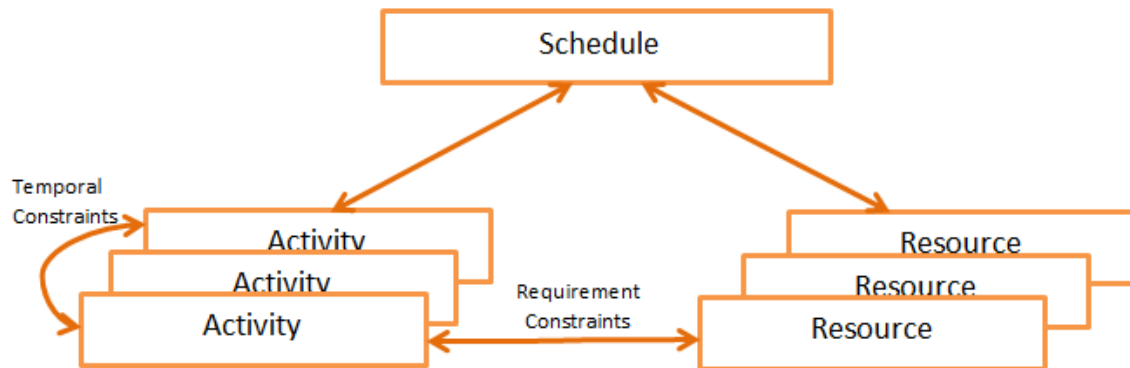
Solving Scheduling Problems

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:

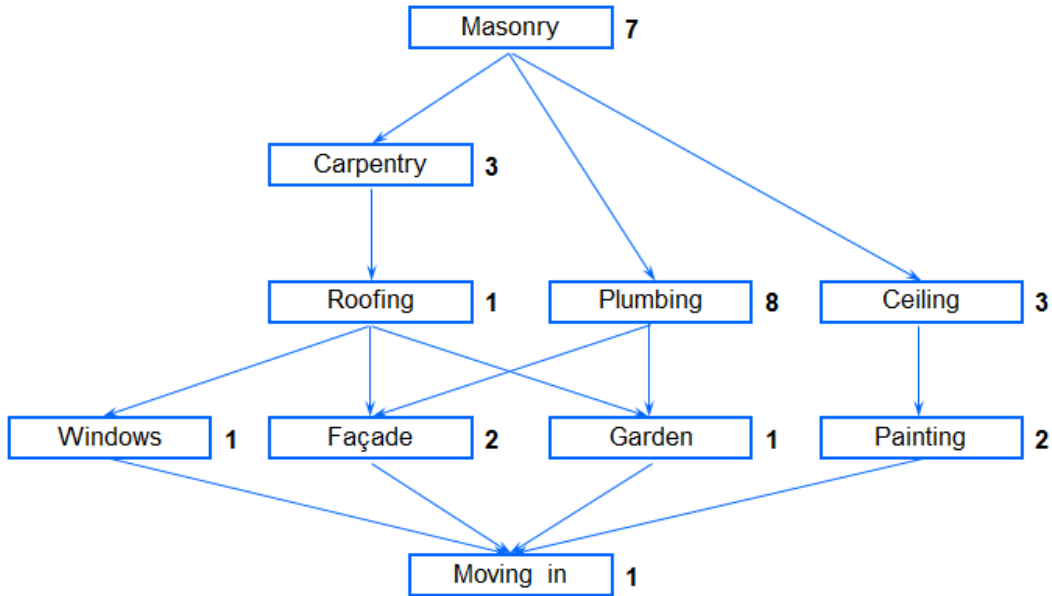


A scheduling problem can be defined in terms of activities and resources. Activities may have an unknown start, duration and end, and may require (or produce) different resources. Resources may have different types (e.g. recoverable like humans or consumable like money) and their limited capacities may 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.

```

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:

Decision ScheduleActivities	
Decisions	Execute Rules
Define Schedule	:= DefineSchedule()
Define Activities	:= DefineActivities()
Define Precedence Constraints	:= DefinePrecedenceConstraints()

We may define schedule in the following table:

DecisionTable DefineSchedule	
ActionSchedule	
Origin	Horizon
0	30

Then we will define activities:

DecisionTable DefineActivities	
ActionAddActivity	
Name	Duration
masonry	7
carpentry	3
roofing	1
plumbing	8
ceiling	3
windows	1
façade	2

garden	1
painting	2
movingIn	1

And finally we will define precedence constraints:

DecisionTable DefinePrecedenceConstraints		
ActionActOperAct		
Activity	Operator	Activity/Day
carpentry	>	masonry
roofing	>	carpentry
plumbing	>	masonry
ceiling	>	masonry
windows	>	roofing
façade	>	plumbing
façade	>	roofing
garden	>	roofing
garden	>	plumbing
movingIn	>	windows
movingIn	>	façade
movingIn	>	garden
movingIn	>	painting

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

```
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
```

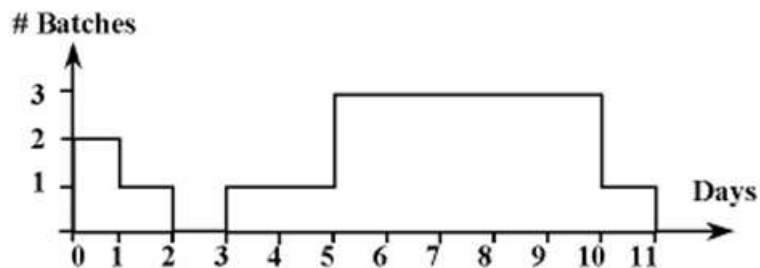
```

=== After Assigning Data:
=== After Posting Constraints:
=== Solve ===
Solution:
masonry[0 -- 7 --> 7)
carpentry[7 -- 3 --> 10)
roofing[10 -- 1 --> 11)
plumbing[7 -- 8 --> 15)
ceiling[7 -- 3 --> 10)
windows[11 -- 1 --> 12)
façade[15 -- 2 --> 17)
garden[15 -- 1 --> 16)
painting[0 -- 2 --> 2)
movingIn[17 -- 1 --> 18)

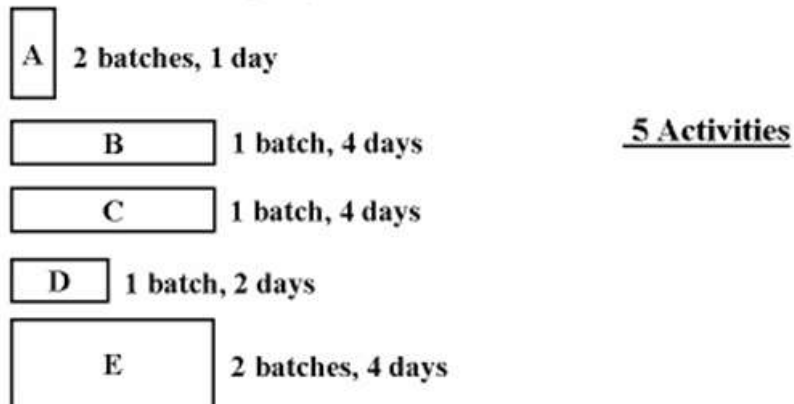
```

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:



Global capacity of the oven



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:

Decision DefineOvenSchedule	
Decisions	Execute Rules
Define Schedule	:= DefineSchedule()
Define Activities	:= DefineActivities()
Define Oven as Recoverable Resource	:= DefineOvenAsResource()
Define Oven Availability	:= SetOvenCapacities()
Define Resource Requirement Constraints	:= ResourceRequirementConstraints()

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

DecisionTable DefineSchedule	
ActionSchedule	
Origin	Horizon
0	11

This table defines all activities:

DecisionTable DefineActivities	
ActionAddActivity	
Name	Duration
A	1
B	4
C	4
D	2
E	4

The resource “Oven” can be created using this table:

DT DefineOvenAsResource		
ActionAddResource		
Name	Type	Max Capacity
Oven		3

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

DT ResourceRequirementConstraints		
ActionActReqResource		
Activity	Required Resource	Required Capacity
A	Oven	2
B	Oven	1
C	Oven	1
D	Oven	1
E	Oven	2

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

DT SetOvenCapacities			
ActionResourceCap			
Resource	From	To	Capacity
Oven	0	1	2
Oven	1	2	1
Oven	2	3	0
Oven	3	5	1
Oven	5	10	3
Oven	10	11	1

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

```
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();
}
```

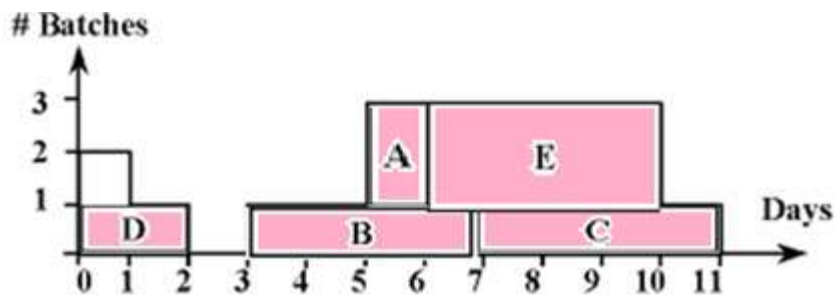
It will produce the following results:

```

*** 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 Availabilityt
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

Now we solve the same problem in 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

```
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:

```
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]
Resource oven: t0[1] t1[1] t2[0] t3[1] t4[1] t5[3] t6[3] t7[3]
t8[3] t9[3] t10[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:

Decision ScheduleActivitiesWithWorker	
Decisions	Execute Rules
Define Schedule	:= DefineSchedule()
Define Activities	:= DefineActivities()
Define Precedence Constraints	:= DefinePrecedenceConstraints()
Define Worker	:= DefineWorker()
Define Resource Requirement Constraints	:= ResourceRequirementConstraints()

We will add a worker using this table:

DecisionTable DefineWorker		
ActionAddResource		
Name	Type	Max Capacity
Worker	Recoverable	1

The next table defines the worker requirement constraints:

DecisionTable ResourceRequirementConstraints		
ActionActReqResource		
Activity	Required Resource	Required Capacity
masonry	Worker	1
carpentry	Worker	1
roofing	Worker	1

plumbing	Worker	1
ceiling	Worker	1
windows	Worker	1
façade	Worker	1
garden	Worker	1
painting	Worker	1
movingIn	Worker	1

Now the same Java launcher will produce the following results:

```

*** Decision ScheduleActivitiesWithWorker ***
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 ScheduleActivitiesWithWorker: Define Schedule
Create RuleScheduler
Decision ScheduleActivitiesWithWorker: Define Activities
Decision ScheduleActivitiesWithWorker: Define Precedence Constraints
Decision ScheduleActivitiesWithWorker: Define Worker
Decision ScheduleActivitiesWithWorker: Define Resource Requirement
Constraints
=== After Assigning Data:
=== After Posting Constraints:
=== Solve ===
Solution:
masonry[0 -- 7 --> 7) requires Worker[1]
carpentry[7 -- 3 --> 10) requires Worker[1]
roofing[10 -- 1 --> 11) requires Worker[1]
plumbing[11 -- 8 --> 19) requires Worker[1]
ceiling[19 -- 3 --> 22) requires Worker[1]
windows[22 -- 1 --> 23) requires Worker[1]
façade[23 -- 2 --> 25) requires Worker[1]
garden[25 -- 1 --> 26) requires Worker[1]
painting[26 -- 2 --> 28) requires Worker[1]
movingIn[28 -- 1 --> 29) requires Worker[1]

```

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:

Decision ScheduleActivitiesWithWorkerBudget	
Decisions	Execute Rules
Define Schedule	:= DefineSchedule()
Define Activities	:= DefineActivities()
Define Precedence Constraints	:= DefinePrecedenceConstraints()
Define Worker & Budget	:= DefineResources()
Define Budget Limitations	:= SetBudgetCapacities()
Define Resource Requirement Constraints	:= ResourceRequirementConstraints()

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

DecisionTable DefineResources		
ActionAddResource		
Name	Type	Max Capacity
Worker	Recoverable	1
Budget	Consumable	30000

The updated resource requirement table will look as follows:

DecisionTable ResourceRequirementConstraints		
ActionActReqResource		
Activity	Required Resource	Required Capacity
masonry	Worker	1
carpentry	Worker	1
roofing	Worker	1
plumbing	Worker	1
ceiling	Worker	1
windows	Worker	1
façade	Worker	1

garden	Worker	1
painting	Worker	1
movingIn	Worker	1
masonry	Budget	1000
carpentry	Budget	1000
roofing	Budget	1000
plumbing	Budget	1000
ceiling	Budget	1000
windows	Budget	1000
façade	Budget	1000
garden	Budget	1000
painting	Budget	1000
movingIn	Budget	1000

And finally the fact that during the first 15 days only \$15,000 are available can be expressed using the following table:

DecisionTable SetBudgetCapacities			
ActionResourceCap			
Resource	From	To	Capacity
Budget	0	15	15000

Now the same Java launcher will produce the following results:

```

*** 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 Joe or Jim
painting  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”:

Decision ScheduleActivitiesWithAlternativeResources	
Decisions	Execute Rules
Define Schedule	:= DefineSchedule()
Define Activities	:= DefineActivities()
Define Precedence Constraints	:= DefinePrecedenceConstraints()
Define Workers	:= DefineWorkers()
Define Resource Requirement Constraints	:= ResourceRequirementConstraints()

The following table defines alternative resources:

DecisionTable DefineWorkers		
ActionAddResource		
Name	Type	Max Capacity
Joe	disjunctive	1
Jack	disjunctive	1
Jim	disjunctive	1

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

DecisionTable ResourceRequirementConstraints		
ActionActReqResource		
Activity	Required Resource	Required Capacity
masonry	Joe Jack	1
carpentry	Joe Jim	1
roofing	Joe Jim	1
plumbing	Jack	1
ceiling	Joe Jim	1
windows	Joe Jim	1
façade	Joe Jack	1
garden	Joe Jim Jack	1
painting	Jack Jim	1
movingIn	Joe Jim	1

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:
=== Solve ===
Solution:
masonry[0 -- 7 --> 7) requires Jack[1]
carpentry[7 -- 3 --> 10) requires Jim[1]
roofing[10 -- 1 --> 11) requires Jim[1]
plumbing[7 -- 8 --> 15) requires Jack[1]
ceiling[7 -- 3 --> 10) requires Joe[1]
windows[11 -- 1 --> 12) requires Jim[1]
façade[15 -- 2 --> 17) requires Jack[1]
garden[15 -- 1 --> 16) requires Jim[1]
painting[0 -- 2 --> 2) requires Jim[1]
movingIn[17 -- 1 --> 18) requires Jim[1]

```

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 several projects projects:

Project “com.openrules.solver“

This project contains an implementation of Rule Solver™:

- Folder “src” with source files:
 - o com.openrules.solver – source code for Rule Solver™
 - o com.openrules.solver.samples - Rule Solver™ examples including “MissManners” that deals with set constrained variables
 - o org.jcp.jsr331.samples - sources with JSR-331 examples
 - o org.jcp.jsr331.scheduler.samples - sources with scheduling examples
- Folder “lib” with supporting jar-files:
 - o jsr331.jar – JSR-331 jar files with 3 open source implementations

- apache/*.jar – Apache Commons jars
- choco/*.jar – jars for Choco’s implementation of the JSR-331
- jacop/*.jar – jars for JaCoP’s implementation of the JSR-331
- constrainer/*.jar – jars for Constrainer’s implementation of the JSR-331
- scheduler.jar – Scheduler’s jar file that also includes all sources
- Folder “rules” with Excel-files for all Rule Solver™ samples:
 - common/Templates.xls – basic OpenRules® templates for defining and solving CSPs
 - common/ScheduleTemplates.xls –OpenRules® templates for scheduling CSPs
 - common/RuleSolver.xls – common Excel files that defines Rule Solver™ environment (called from all other Excel files)
 - <Name>.xls – various Rule Solver™ examples (without decisions)

Project “openrules.config“

This project contains standard OpenRules® jars (in the folder “lib”) and decision templates.

Decision Projects

There are many sample projects such as DecisionHelloCP, DecsionLoanCP, DecisionScheduleActivities, and others described above.

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](#).