BPM Voices: Using business rules in the cloud to solve Sudoku, Part 1: Implementing the rules application

Rajesh Rao
Decision Management Solution Architect
IBM

Skill Level: Introductory

Date: 06 Jun 2012

This series presents a hybrid approach that combines business rules and heuristic depth-first search to solve Sudoku puzzles. This approach emulates the methodology humans use to solve Sudoku. Rules built using WebSphere® Operational Decision Management apply forward-chaining logic; when this fails to find the solution, the heuristic search with backtracking emulates the trial-and-error approach. In Part 2 of this series, we'll outline the architecture and process used to embed the ruleset into a web application hosted on a public cloud.

View more content in this series

Before you begin

This column is for WebSphere Operational Decision Management developers, and it is assumed that the reader has a fairly good understanding of the product from a developer's perspective. Refer to the Resources section for links to prerequisite knowledge for completing the steps in this column. The product version I used for this column is WebSphere Operational Decision Management, V7.5, specifically the Rule Designer.

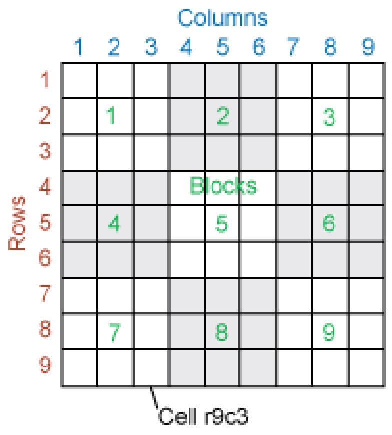
Novice developers who have more of a procedural programming background can benefit by the illustration of how rules efficiently, concisely and implicitly iterate over the rows and columns of a Sudoku grid. More advanced developers can benefit from the discussions relating to rule task partitioning for efficiency and how rules can be used as part of a hybrid approach, which has broader applicability in other constraint satisfaction problems.

Introduction

A few years ago I visited my grandparents' home, which doubles as a favorite vacation home for several of my young rambunctious cousins, expecting to barge into a rollicking game of cricket or some such activity, only to find my family members silently huddled in a room, engrossed in putting pencil to paper, as though they were in the middle of an intense exam. It turned out that they were all solving Sudoku. That was my first exposure to the beguiling power of the now enormously popular Sudoku. And in the interest of full disclosure, I must confess to now being an avid fan too.

A Sudoku puzzle (shown in Figure 1), typically with 9 rows and 9 columns for a total of 81 cells, is a simple puzzle with a simple objective: fill this grid with numbers from 1 through 9 such that each row, column, and 3x3 block within the larger grid contains all the numbers from 1 through 9; which means that no number can repeat in a row, column or block. A puzzle provides starting values for some of the cells. The remaining empty cells are to be filled in by the solver.

Figure 1. A Sudoku puzzle



Each cell in the grid is identified by the row and column associated with it. For example, the cell in the 9th row, 3rd column is written as *r9c3*.

It has been shown that the number of classic 9×9 Sudoku solution grids is very large -- approximately 6.67×10^{21} -- and the general problem of solving Sudoku puzzles is NP-complete. The joy in solving Sudoku arises from arriving at the right one of these possible solutions through a series of incremental logical steps, each progressively getting the solver closer to the solution. Each of these logical steps involves finding patterns among the populated cells. Filling in a cell further constrains the other empty cells. This forward-chaining approach makes it a classic candidate for emulation with a forward-chaining rule engine, such as WebSphere Operational Decision Management, where the assertion of new knowledge causes all relevant rules to fire exhaustively, thereby progressing from the original state to the goal state in a step-by-step manner.

However, there are times when human solvers get stuck and can't make any progress. At this point, they may resort to a trial and error methodology where they have to guess the value of a cell and then proceed with the knowledge that they may need to backtrack and change this guessed value and all values derived since.

The need for guessing is inversely proportional to the amount of knowledge in the knowledge base available for forward chaining. The efficiency of this trial and error approach can vary greatly depending on which cell is chosen to hazard a guess on.

In building our sample rule application, this human approach is emulated in the rich tradition of AI. The goals of the rule application are:

1. Use heuristics that are commonly used by Sudoku solvers. Therefore, we eschew the more esoteric heuristics found in Sudoku literature, such as X-Wings and Swordfish.
2. Apply trial and error only as the last resort.
3. Provide a step-by-step facility to explain why a Sudoku cell is populated in a certain way.

Note: There are several approaches for solving Sudoku, and this column should not be misconstrued to imply that a rule-based approach is the only approach or even the best approach. For instance, a constraint programming representation for solving Sudoku, using for example the IBM® ILOG Operational Decision Manager Enterprise platform, can be very concise and declarative.

Rule implementation

Business rules effectively and efficiently capture the heuristic knowledge that humans use to solve Sudoku. This section outlines this heuristic knowledge and the corresponding business rule implementation.

Nine Heuristics

As adapted from Bob Hanson's Sudoku Assistant, we identify nine heuristics that will be implemented using business rules to solve Sudoku puzzles. In addition to the row, column and block of a cell, these heuristics refer to a *candidate*, which is nothing but a possible value that can be placed in a cell. Therefore, `candidate k` refers to a possible value of `k` in a cell, where `k` is a digit between 1 and 9.

- **Heuristic 1: Heuristic of exclusion**
  If a cell has value k, then no other cell in the same row, column or block can be k.
- **Heuristic 2: Heuristic of naked singles**
  If a candidate k is possible in a cell and no other candidates are possible in that cell, then that cell must be k.
- **Heuristic 3: Heuristic of hidden singles**
  When a candidate k is possible in only a single cell of a row, column, or block, then that cell must be k.
- **Heuristic 4: Heuristic of locked candidate in row/column**
  If a candidate k is possible in a certain intersection of row/column and block but is not possible elsewhere in that row/column, then it is also not possible elsewhere in that block.

- **Heuristic 5: Heuristic of locked candidate in block**
  If a candidate k is possible in a certain intersection of row/column and block but is not possible elsewhere in that block, then it is also not possible elsewhere in that row/column.
- **Heuristic 6: Heuristic of naked pairs**
  If 2 candidates are possible in a set of 2 cells of a given row/column, and no other candidates are possible in those 2 cells, then those 2 candidates are not possible elsewhere in that row/column.
- **Heuristic 7: Heuristic of hidden pairs**
  If 2 candidates are possible in a set of 2 cells of a given row/column, and those 2 candidates are not possible elsewhere in that row/column, then no other candidates are possible in those 2 cells.
- **Heuristic 8: Heuristic of naked trio**
  If 3 candidates are possible in a set of 3 cells of a given block and no other candidates are possible in those 3 cells, then those 3 candidates are not possible elsewhere in that block.
- **Heuristic 9: Heuristic of hidden trio**
  If 3 candidates are possible in a set of 3 cells of a given block, and those 3 candidates are not possible elsewhere in that block, then no other candidates are possible in those 3 cells.

The Business Object Model (BOM)

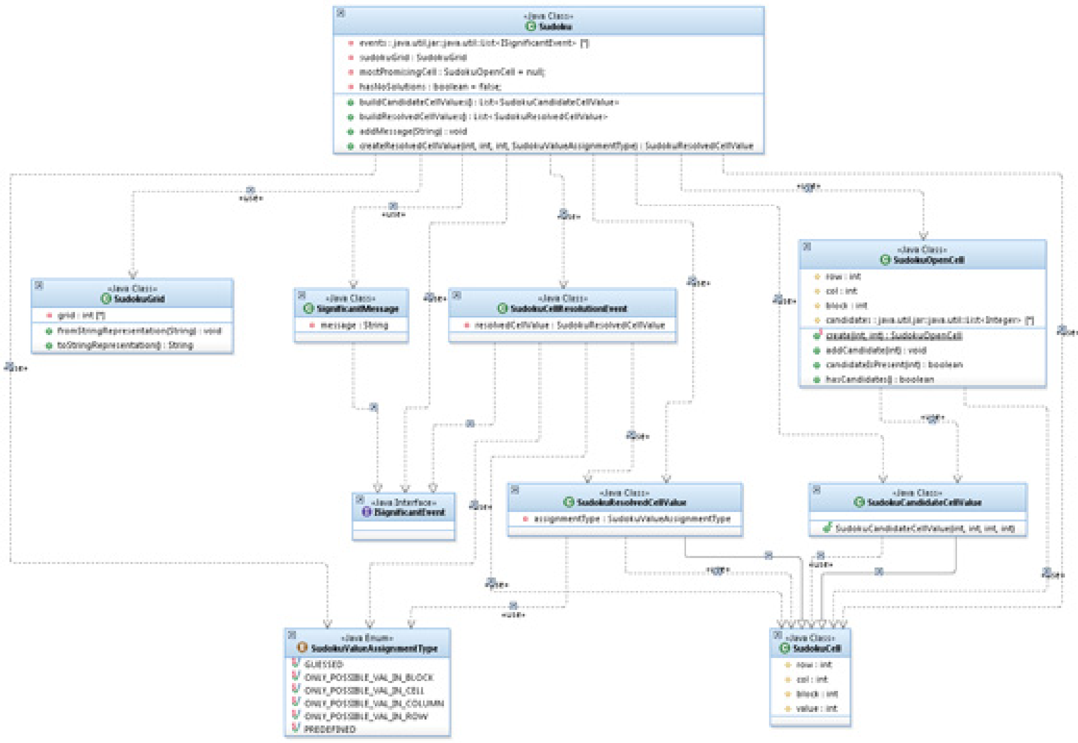
Before embarking on building the rules that embody these nine heuristics, it's important to define the data structures that form the vocabulary for defining the rules. Some key classes jump out from the analysis of the heuristics in the previous section:

1. Sudoku grid: This is a 2-dimensional array of integers that represents a Sudoku board.
2. Candidate cell value: This object represents a potential candidate value for a cell.
3. Resolved cell value: This object represents a resolved cell with a committed value assigned to it. An assignment type is associated with this object to identify the rule that assigned the value.

These objects are shown in the class diagram in Figure 2. The class named `Sudoku` is a higher-level container class that represents the input and output to the rule engine. As noted earlier, in addition to solving for the cell values, the rule engine has the additional responsibility of providing an explanation facility. During rule execution, all significant messages are added to the Sudoku container object. These significant messages may be ad hoc messages or cell resolution events. All important events are captured in this message list.

In the event that the rules do not arrive at a solution, rules enable efficient trial and error processing by identifying the most promising cell to make a guess on. This is encapsulated in the `SudokuOpenCell` class shown below. Notice that the Sudoku container has an attribute called the `mostPromisingCell` of type `SudokuOpenCell` that is set if necessary by the rule engine.

Figure 2. Execution Object Model


«Java Class»
Sudoku
events : java.util.jar:java.util::List<ISignificantEvent> [*]
sudokuGrid : SudokuGrid
mostPromisingCell : SudokuOpenCell = null;
hasNoSolutions : boolean = false;
buildCandidateCellValues() : List<SudokuCandidateCellValue>
buildResolvedCellValues() : List<SudokuResolvedCellValue>
addMessage(String) : void
createResolvedCellValue(int, int, int, SudokuValueAssignmentType) : SudokuResolvedCellValue
«Java Class»
SudokuGrid
grid : int [*]
fromStringRepresentation(String) : void
toStringRepresentation() : String
«Java Class»
SignificantMessage
message : String
«Java Class»
SudokuCellResolutionEvent
resolvedCellValue : SudokuResolvedCellValue
«Java Class»
SudokuOpenCell
row : int
col : int
block : int
candidates : java.util.jar:java.util::List<Integer> [*]
create(int, int) : SudokuOpenCell
addCandidate(int) : void
candidateIsPresent(int) : boolean
hasCandidates() : boolean
«Java Interface»
ISignificantEvent
«Java Class»
SudokuResolvedCellValue
assignmentType : SudokuValueAssignmentType
«Java Class»
SudokuCandidateCellValue
SudokuCandidateCellValue(int, int, int, int)
«Java Enum»
SudokuValueAssignmentType
GUESSED
ONLY_POSSIBLE_VAL_IN_BLOCK
ONLY_POSSIBLE_VAL_IN_CELL
ONLY_POSSIBLE_VAL_IN_COLUMN
ONLY_POSSIBLE_VAL_IN_ROW
PREDEFINED
«Java Class»
SudokuCell
row : int
col : int
block : int
value : int
«use»

(See a larger version of Figure 2.)

Rule representation

There are essentially two types of rules for solving Sudoku: rules that eliminate potential candidates and rules that assign a number to a cell. These rules make the assumption that all the applicable candidates and resolved cell values are in working memory. The elimination rules remove candidates from working memory, while the assignment rules insert a resolved cell value into working memory apart from updating the Sudoku grid. Use of the working memory in the rule engine allows the Rete network to become aware of changes to the state of the Sudoku grid and to appropriately trigger other rules when that happens.

Assignment rules

The heuristic of naked singles and the heuristic of hidden singles described earlier are the only assignment rules. Their implementation is quite straightforward. For example, the heuristic of naked single that assigns a value to a cell if it is the only candidate in a block is shown in the following listing. The rule condition in lines 4-7 checks whether there is only one candidate for a value in a block. If that condition is satisfied, then the action in lines 9 and 10 retracts the candidate, assigns the single value to the cell, creates a resolved cell object, and inserts it into working memory.

```
definitions
set 'candidateCell' to a candidate cell value  ;
if
the number of candidate cell values
  where the block of each candidate cell value equals the block of candidateCell
  and the value of each candidate cell value  equals the value of candidateCell ,
 equals 1
then
retract candidateCell ;
  insert object : create a resolved cell value using candidateCell ,
assignment type: ONLY_POSSIBLE_VAL_IN_BLOCK ;
```

This rule checks for naked singles in a block. Versions of this rule that check for singles in rows and columns are also added.

The heuristic of hidden singles checks whether there is only one candidate remaining for a cell, and is implemented by the rule shown in the following listing.

```
definitions
set 'candidateCell' to a candidate cell value ;
if
the number of candidate cell values
  where the row of each candidate cell value equals the row of candidateCell
  and the col of each candidate cell value equals the col of candidateCell ,
 equals 1
then
retract candidateCell ;
insert object : create a resolved cell value using candidateCell ,
assignment type: ONLY_POSSIBLE_VAL_IN_CELL ;
```

Elimination rules

Elimination rules remove potential candidates and can range in complexity from simple to very complex. On the simple end, we have the heuristic of exclusion, which removes candidates from the row, column and block of a resolved cell. The heuristic of exclusion as applied to rows is shown below, with similar versions to eliminate candidates from columns and blocks.

```
definitions
 set 'resolvedCell' to a resolved cell value ;
 set 'candidateCell' to a candidate cell value
  where the row of this candidate cell value  equals the row of
resolvedCell
  and the value of this candidate cell value  equals the value of
resolvedCell ;
then
 retract candidateCell ;
```

Higher up in the complexity scale are the rules relating to locked candidates. For instance, as you may recollect, the heuristic of locked candidate in block states:

```
If a candidate k is possible in a certain intersection of
row and block but is not possible elsewhere in that block,
then it is also not possible elsewhere in that row.
```

To restate, if there are no candidates in a block other than in a single row (that is, the candidate is locked to that block in that row), then eliminate that candidate from other cells in that row. The rule implementation is shown in the following listing. Lines 3-6 bind the other candidates that are in a row that need to be eliminated when the condition in lines 8-12 is satisfied. In the rule action (line 14), we add this elimination as a significant event to display as part of the explanation facility.

```
1. definitions
2. set 'candidateCell1_val1' to a candidate cell value ;
3. set 'othercandidateCellValue' to a candidate cell value
4.   where the block of this candidate cell value is not the block of
candidateCell1_val1
5.   and the row of this candidate cell value is the row of candidateCell1_val1
6.   and the value of this candidate cell value is the value of
candidateCell1_val1 ;
7. if
8. the number of candidate cell values
9.   where the row of each candidate cell value is not the row of
candidateCell1_val1
10.  and the value of each candidate cell value is the value of
candidateCell1_val1
11.  and the block of each candidate cell value is the block of
candidateCell1_val1 ,
12.     equals 0
13. then
14. add "Found locked cell value of " + the value of candidateCell1_val1  + "
in block "
+ the block of candidateCell1_val1 + ". Removing this value from the row " +
the row
of othercandidateCellValue  to the messages of 'the sudoku' ;
15. retract othercandidateCellValue  ;
```

Further up in the complexity chain are the rules that relate to naked pairs and hidden pairs. To refresh your memory, the heuristic of naked pair states:

```
If 2 candidates are possible in a set of 2 cells of a given row,
and no other candidates are possible in those 2 cells, then those 2
candidates are not possible elsewhere in that row.
```

The rule implementation is shown in the following listing. `Cell1` and `cell2` are the two cells with the naked pair. Line 7 in the definitions section ensures that the two cells are in the same row. Each of these cells has only two candidates, as checked in the rule condition between lines 17-25. All the other candidates bound in lines 13-15 are then eliminated by retracting them from working memory (line 28).

```
1. definitions
2. set 'cell1_candidate1' to a candidate cell value ;
3. set 'cell1_candidate2' to a candidate cell value
4.   where the row of this candidate cell value is the row of cell1_candidate1
5.      and the col of this candidate cell value is the col of cell1_candidate1 ;
6. set 'cell2_candidate1' to a candidate cell value
7.   where the row of this candidate cell value is the row of cell1_candidate1
8.      and the value of this candidate cell value is the value of cell1_candidate1 ;
9. set 'cell2_candidate2' to a candidate cell value
10.    where the row of this candidate cell value is the row of cell2_candidate1
11.      and the col of this candidate cell value is the col of cell2_candidate1
12.       and the value of this candidate cell value is the value of
cell1_candidate2 ;
```

```
13. set 'otherCandidate' to a candidate cell value
14.    where the row of this candidate cell value is the row of cell1_candidate1
15.      and the value of this candidate cell value is the value of
cell1_candidate1 ;
16. if
17. the number of candidate cell values
18.    where the row of each candidate cell value is the row of cell1_candidate1
19.      and the col of each candidate cell value is the col of cell1_candidate1 ,
20.    equals 2
21. and
22. the number of candidate cell values
23.   where the row of each candidate cell value is the row of cell2_candidate1
24.      and the col of each candidate cell value is the col of cell2_candidate1 ,
25. equals 2
26. then
27. add "Rule - naked pair: Removing " + the value of otherCandidate   + " from row "
+ the row of otherCandidate  to the messages of 'the sudoku' ;
28. retract otherCandidate  ;
```

The most complex rules are the ones relating to trios. The rule of hidden trio asserts that when three candidates are possible in a certain set of three cells all in the same block, and those three candidates are not possible elsewhere in that same block, then no other candidates are possible in those cells.

This is implemented in the following listing. In this rule, we bind to three candidates in each of three cells for a total of nine bindings, as seen in lines 2-30. The rule condition (lines 36-52) ensures that there are only three candidates for each of these three cells, implying that there are no other candidates apart from the ones already bound in the definitions section. When that is true, all other candidates (bound on line 31) for `cell1` are removed from working memory (in line 55). Note that we do not explicitly need to remove candidates from `cell2` and `cell3` because the Rete algorithm ensures that the rule will match on these cells as `cell1` in other activations of this rule.

```
1. definitions
2. set 'cell1_candidate1' to a candidate cell value ;
3. set 'cell1_candidate2' to a candidate cell value
4.   where the row of this candidate cell value is the row of cell1_candidate1
5.       and the col of this candidate cell value is the col of
cell1_candidate1 ;
6. set 'cell1_candidate3' to a candidate cell value
7.   where the row of this candidate cell value is the row of cell1_candidate1
8.       and the col of this candidate cell value is the col of
cell1_candidate1 ;
9. set 'cell2_candidate1' to a candidate cell value
10.   where the block of this candidate cell value is the block of cell1_candidate1
11.       and the value of this candidate cell value is the value of
cell1_candidate1 ;
12. set 'cell2_candidate2' to a candidate cell value
13.   where the row of this candidate cell value is the row of cell2_candidate1
14.       and the col of this candidate cell value is the col of cell2_candidate1
1       and the value of this candidate cell value is the value of
cell1_candidate2 ;
16. set 'cell2_candidate3' to a candidate cell value
17.   where the row of this candidate cell value is the row of cell2_candidate1
18.       and the col of this candidate cell value is the col of cell2_candidate1
19.       and the value of this candidate cell value is the value of
cell1_candidate3 ;
20. set 'cell3_candidate1' to a candidate cell value
```

```
  where the block of this candidate cell value is the block of cell1_candidate1
      and the value of this candidate cell value is the value of cell1_candidate1 ;
set 'cell3_candidate2' to a candidate cell value
  where the row of this candidate cell value is the row of cell3_candidate1
      and the col of this candidate cell value is the col of cell3_candidate1
      and the value of this candidate cell value is the value of cell1_candidate2 ;
set 'cell3_candidate3' to a candidate cell value
  where the row of this candidate cell value is the row of cell3_candidate1
      and the col of this candidate cell value is the col of cell3_candidate1
      and the value of this candidate cell value is the value of cell1_candidate3 ;
set 'otherCandidate' to a candidate cell value
  where
    the row of this candidate cell value is the row of cell1_candidate1
      and the col of this candidate cell value is the col of cell1_candidate1 ;
if
the number of candidate cell values
  where
    the block of each candidate cell value is the block of cell1_candidate1
    and the value of each candidate cell value is the value of cell1_candidate1 ,
  equals 3
and
the number of candidate cell values
  where
    the block of each candidate cell value is the block of cell1_candidate2
        and the value of each candidate cell value is the value of cell1_candidate2 ,
    equals 3
and
the number of candidate cell values
    where
    the block of each candidate cell value is the block of cell1_candidate3
    and the value of each candidate cell value is the value of cell1_candidate3 ,
    equals 3
then
   add "Rule - hidden trio: Removing " + the value of otherCandidate   +
" from block "
+ the block of otherCandidate  to the messages of 'the sudoku' ;
   retract otherCandidate  ;
```

This rule only catches perfect hidden trios, where all three candidates are present in the three cells. However, even if one of these cells has only two of the candidates, then it can still qualify as a hidden trio. Considering the performance cost for these complex rules, we choose not to implement versions of this rule to handle imperfect trios. This trade-off is applied to higher order subsets too. For example, the performance cost to implement rules to handle "quads" in four cells is not justified for the extremely rare configuration in which it is applicable and helpful. In those rare cases, the trade-off tips in favor of a trial and error approach.

Guessing Rules

When a solution is not found through logical application of rules, then the solver needs to resort to trial and error. The efficiency of this trial and error approach greatly depends on which cell is chosen for guessing. Rules apply a simple heuristic to determine this cell -- pick the cell with the least number of remaining candidates.

Rules execute the following steps in identifying the most promising cell:

1. Create open cells and insert them into working memory.
2. Add candidate values to these open cells.
3. Pick the open cell with the least number of candidates as the most promising cell.

The rule to create the open cells illustrates how we can implicitly iterate over a list of values and is listed below. Open cells are created for all cells that don't have a corresponding resolved cell value (as checked in lines 5-7).

```
1. definitions
2. set rowNum to a number in { 1,2,3,4,5,6,7,8,9} ;
3. set colNum to a number in { 1,2,3,4,5,6,7,8,9} ;
4. if
5.  there is no resolved cell value
6.     where the row of this resolved cell value is rowNum
7.     and  the col of this resolved cell value is colNum,
8. then
9.  insert object : create an open cell for row rowNum and column colNum ;
```

Likewise, the rule to add candidate values to the open cells is very simple, and is listed below. This rules matches on all candidates in working memory and adds them to the appropriate open cell.

```
1. definitions
2. set openCell to a sudoku open cell ;
3. set candidate to a candidate cell value
4.   where the row of this candidate cell value is the row of openCell
5.   and the col of this candidate cell value is the col of openCell ;
6. then
7.   add the value of candidate  to the candidates of openCell ;
```

The rule that applies the heuristic of minimum number of candidates is listed below. The rule condition (line 6-7) checks to ensure that there are no open cells with fewer candidates and then sets that open cell as the most promising cell for guessing.

```
1. definitions
2. set openCell to a sudoku open cell
3.   where the number of candidates of this sudoku open cell is at least 1 ;
4. if
5. the most promising cell of 'the sudoku' is null and
6. there is no sudoku open cell where
7.    the number of candidates of this sudoku open cell is less than the number of
candidates of openCell ,
8. then
9. set the most promising cell of 'the sudoku' to openCell ;
```

If an open cell is found with no candidates, then the Sudoku has no solution. This is implemented by the rule shown below.

```
definitions
set openCell to a sudoku open cell ;
if
it is not true that openCell has candidates
then
 make it true that 'the sudoku' has no solution ;
```

Validation Rules

To round off the discussion of the rules in the application, let's look at the validation rules that check for inconsistencies in the Sudoku grid. These validation rules check if two cells in the same row or column or block have the same assigned value, which would be illegal. An example is shown in the listing below. This rule sets an indicator that the Sudoku has no solution (line 8) if a particular value exists in two resolved cells in the same block (lines 2-5).

```
1. definitions
2. set 'resolvedCell1' to a resolved cell value ;
3. set 'resolvedCell2' to a resolved cell value
4.    where the block of this resolved cell value  equals the block of resolvedCell1
5.    and the value of this resolved cell value equals the value of resolvedCell1 ;
6. then
7. print "Block conflict in block " + the col of resolvedCell1 + " for value " + the
value of resolvedCell2 ;
8. make it true that 'the sudoku' has no solution ;
```

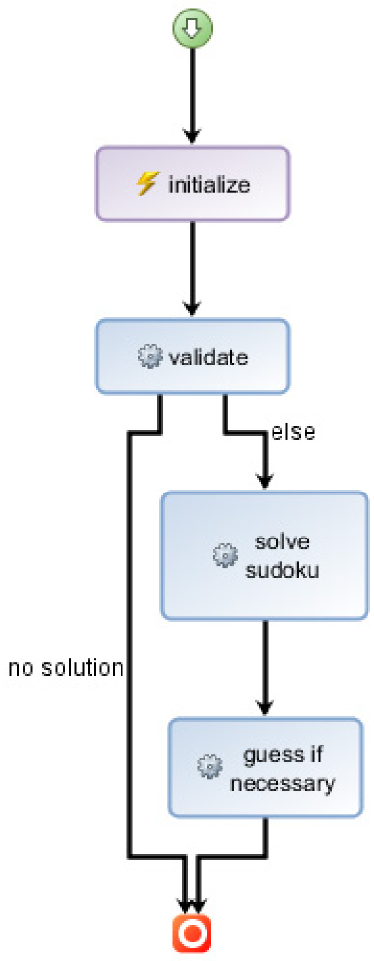
This gives a flavor of all the rules used in the application. For a complete listing of the rules, please download the rule report from the Downloads section.

Rule processing and optimization

The main ruleflow, which orchestrates the execution of rules, is shown in Figure 3. The execution of rules goes through the following steps:

1. Iniitialization, where the candidates and resolved cells are created and inserted into the working memory of the rule engine.
2. Validation, where the resolved cells are checked to ensure that there are no inconsistencies.
3. Solving, where rules are applied to solve the Sudoku.
4. Guessing, if no solution was found.

Figure 3. Main ruleflow



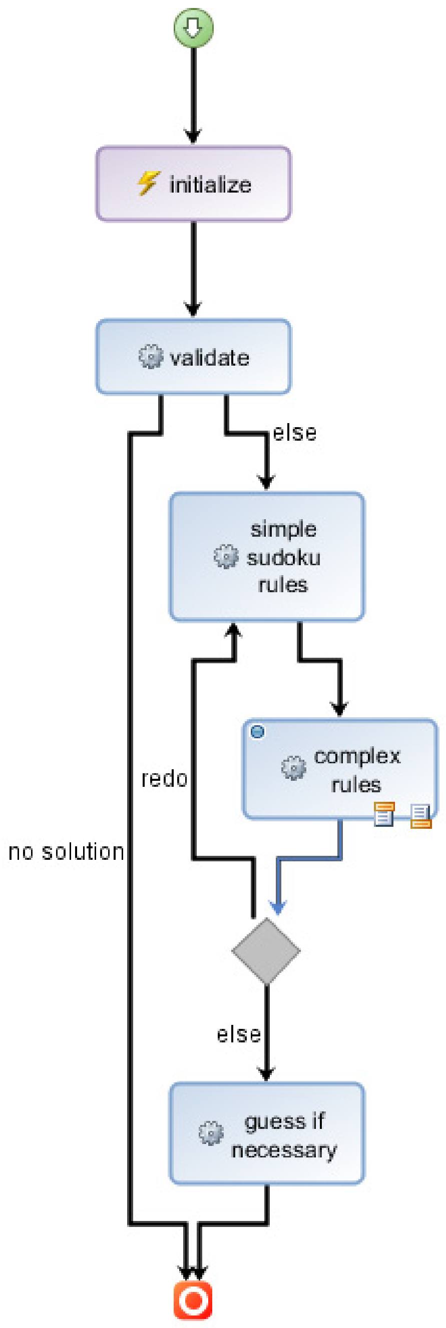
Note that initialization is an action task unlike the other tasks, which are rule tasks. The action task creates resolved cells and candidates from the input Sudoku, as shown in the following listing.

```
List resolvedCells = sudoku.buildResolvedCellValues();
List possibleCells = sudoku.buildCandidateCellValues();

int i;
for (i=0; i<resolvedCells.size(); i++) {
 context.insert(resolvedCells.get(i));
}
for (i=0; i<possibleCells.size(); i++) {
 context.insert(possibleCells.get(i));
}
context.insert(sudoku);
```

Once the candidates and resolved cells are put into working memory, the rules to solve Sudoku can kick into action. However, there is a problem with this rule processing. When there are a lot of candidates, the Rete network in the rule engine gets overburdened and the processing time is unacceptably long. However, if the complex rules are removed, the solution is found in a fraction of a second. This is understandable because of the amount of combinatorial Rete network activity generated by the complex rules patterns. For example, if there are 500 candidates in working memory, and a hidden trio rule matches across 10 different candidates, there are roughly 500^{10} potential permutations that the Rete network has to sort through!

The complex rules pattern matching cost goes up exponentially with the number of candidates in working memory. To reduce the processing imprint of these complex rules, we can separate them out into a different rule task that processes only those candidates that have not already been eliminated by the simple rules. Since the simple rules drastically reduce the number of candidates, this change has a major positive impact on performance. The ruleflow with this optimization is depicted in Figure 4, where the complex rules are executed in a separate rule task after the completion of the simple Sudoku rules.

Figure 4. Ruleflow with optimization



As you can see, one drawback to this optimization is the added complexity to the ruleflow. Not only are the rules split into simple and complex rules, but there is also a loop back from the complex rules to the simple rules. This is because the elimination of a candidate by a complex rule may trigger a whole series of other inferences by the simple rules. Therefore, if any of the complex rules fire, the simple rules will need to be invoked again. However, this added complexity is worthwhile because it results in a couple of orders of magnitude in performance improvement.

To implement this, a variable is set as an action of the complex rules, which is then used in the rule flow transition logic. Line 5 in the snippet below, which is an excerpt of the action part of a complex rule, illustrates this.

```
1. If
2. …
3. then
4. …
5. set 'redo flag' to true ;
```

Depth-first search

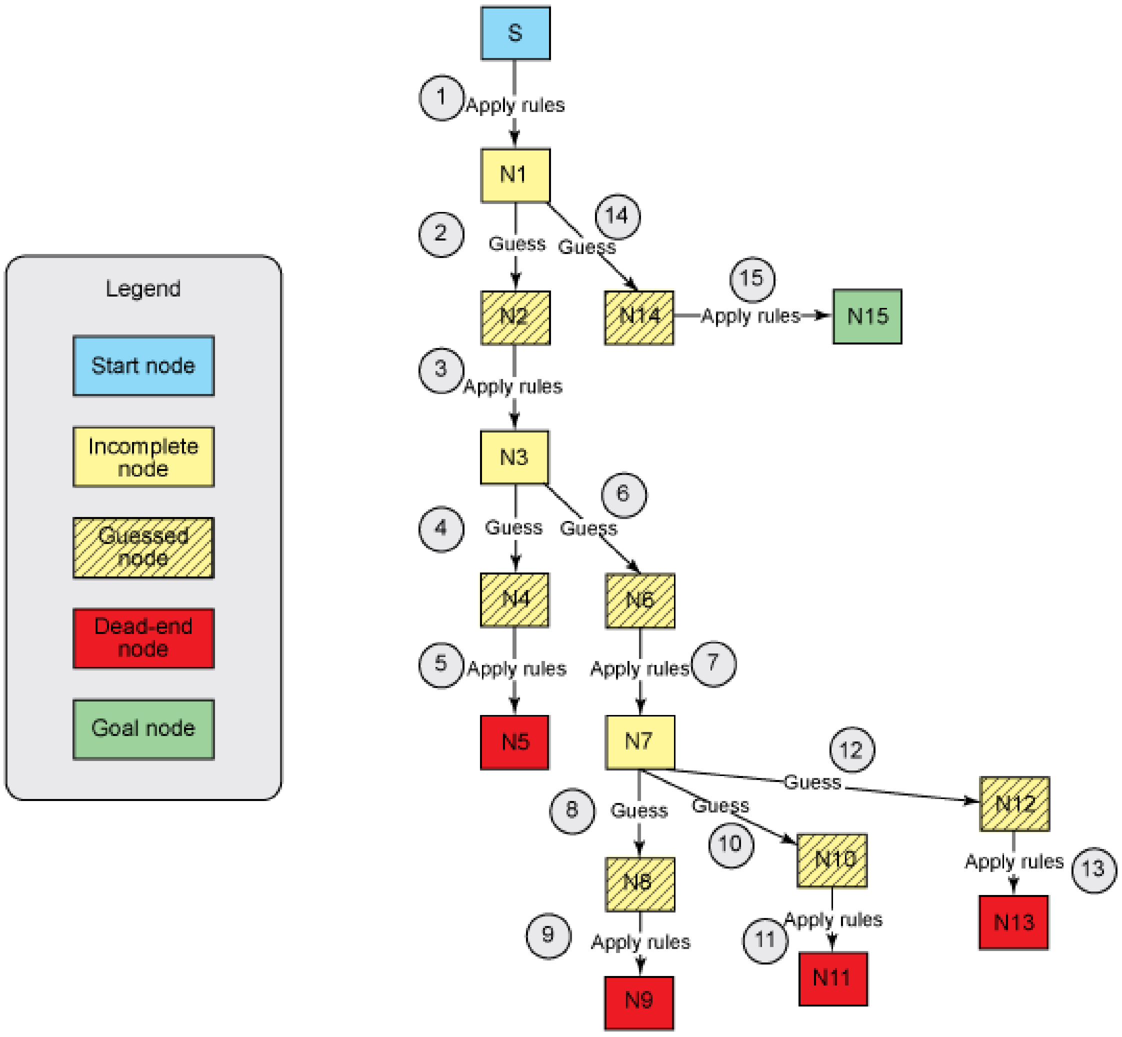
The business rules described in the previous section can solve even many difficult Sudoku configurations, but not all. There are Sudoku configurations where none of these business rules apply, in which case the rule engine has to resort to trial and error, or guessing. The process of trial and error is jointly handled by the rule engine and a heuristic depth-first solver.

Depth-first search is an algorithm for searching a tree structure starting from an initial node and exploring as far as possible along a branch until either the goal node or a dead end is reached before stopping or backtracking. This in an uninformed search and does not take the goal into account in selecting a node from a list or candidate nodes. A heuristic depth first search, on the other hand, uses some heuristic function to determine which of the nodes is the most promising.

A node in our search tree is nothing but a Sudoku configuration. The depth-first search algorithm progresses from one node in the search tree to the next by guessing a value for an empty cell or executing the rule engine. Execution of the rule engine can result in one of three situations: 1) rule engine finds a solution; 2) rule engine determines that there is an inconsistency or no solution exists; and 3) rule engine stops with an incomplete grid because no more rules apply. In the first case, the depth-first search solver stops; in the second case, the solver backtracks; and in the third case, the solver makes a guess to get to the next node, unless the maximum depth has been reached, in which case it backtracks. Depth here refers to the number of guesses already made along the search branch.

An example of this hybrid depth-first search solver is depicted in Figure 5. In this example, the solver invokes the rule engine with the starting node S. The rule engine applies business logic and fills in several cell values resulting in node N1. However, the solution has not been reached with the application of the business rules, therefore the solver resorts to trial and error. It makes a guess on one of the empty cells of node N1 resulting in node N2. This node is then processed through the rule engine resulting in N3. Again, since this is not the goal state, the next node is obtained by making a guess on N3 resulting in node N4. When the rule engine is invoked on N4, it reports back that there are no possible solutions for this configuration; in other words, it is a dead end. Now the solver backtracks to N3 and applies a different guess (step 6) on the cell to result in N6. Following along the steps from 6-13, you can see that all the nodes are dead ends. Therefore, the solver now backtracks all the way back to node N1 and applies the next possible value for the guessed cell (step 14). When this configuration is passed to the rule engine, it finds the solution N15.

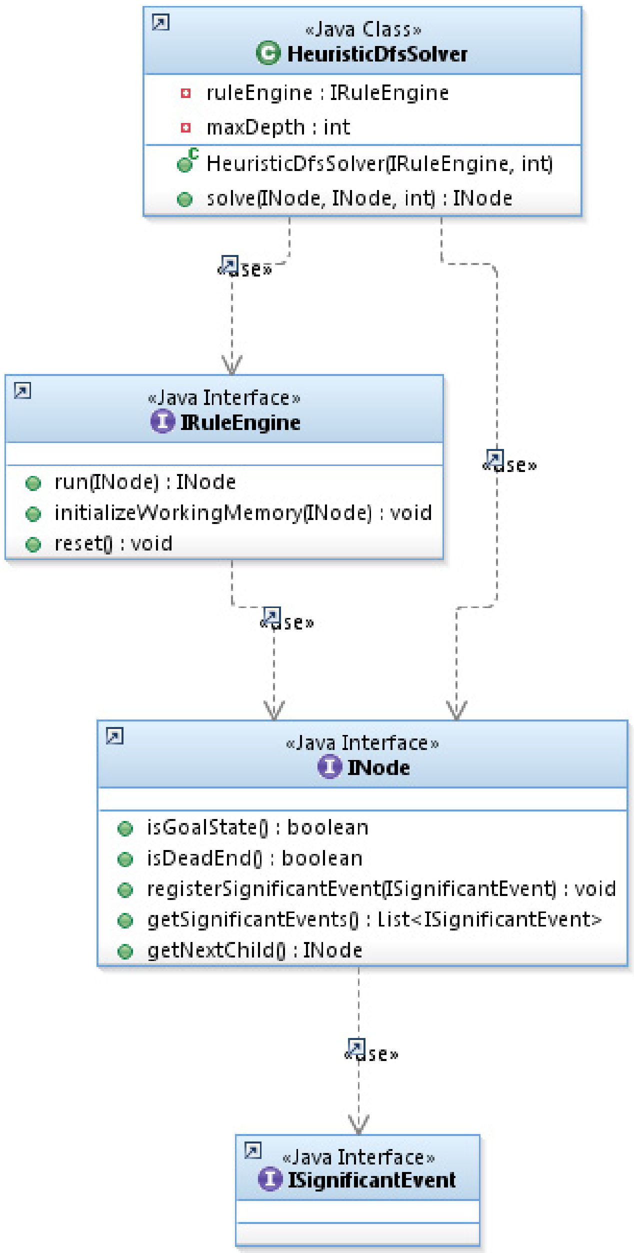
Figure 5. Depth-first search with rules



A blind depth-first search strategy makes a guess by choosing the first available candidate for the first empty cell. This means that an empty cell containing nine candidates may be picked instead of a different cell with only two candidates. When picking the cell with nine possible values, it is statistically likely that the first four lead to dead ends. When the selected cell has only two candidates, the likelihood of the first guess being right is 0.5 -- much higher than the 1/9 of the first case. Therefore, heuristics that pick the most promising cell can have a huge impact on the performance of the depth-first search. We use the rule engine to identify the most promising cell when it reaches an incomplete and consistent board configuration where no more rules apply. As seen in the previous section, business rules apply a simple heuristic to determine the most promising cell -- pick the cell with the least number of potential candidates.

This hybrid approach of combining business rules and algorithmic search with backtracking has applicability beyond solving Sudoku to other constraint satisfaction problems, such as product configuration. Recognizing this fact, a generic algorithm has been devised that can use any rule engine and work with any search tree. Figure 6 depicts the structures used to represent the key elements used. Each node in the search tree is represented by an INode interface, which has methods to determine whether the node is a goal state or dead end. It also has a method to determine the next child node to consider after backtracking. This next child node, in our case, is the result of applying a guess. The rule engine has a run method that takes an INode as an argument and returns the updated node after applying all the business rules.

Figure 6. Hybrid depth-first search model



The algorithm used to apply the hybrid approach takes a recursive form and is listed below.

```
solve (node, parent, depth)
if (depth > MAX_DEPTH) return no_solution
node = run rules (node)
if (node is goal state), return node
else
if (node is dead end) return no_solution
else // partial solution found
while (there is a get next best_child and solution_node is not found)
    solution_node =  solve (best_child, node, depth + 1)
return solution_node
```

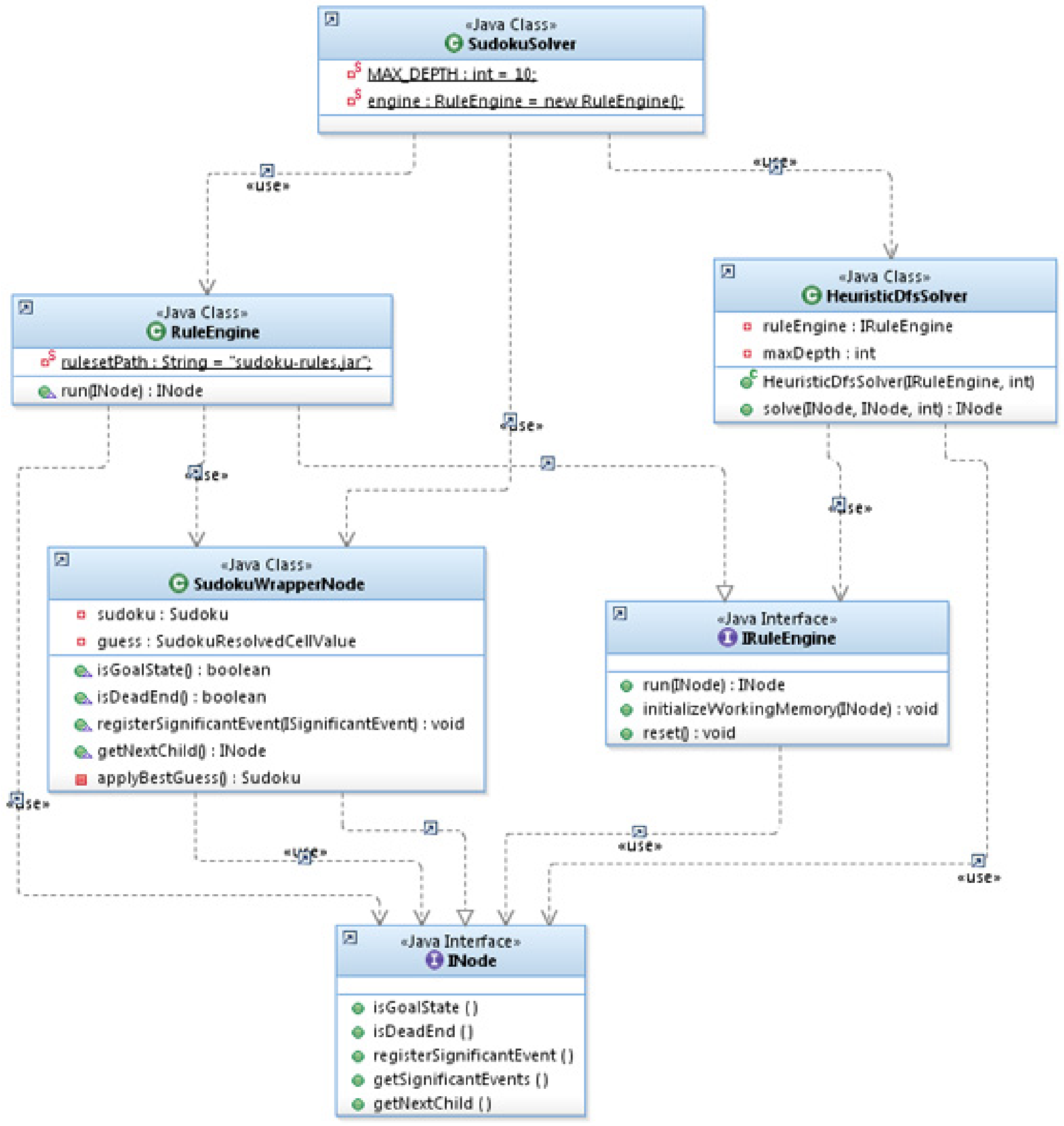
The corresponding Java snippet for the HeuristicDfsSolver is:

```
public INode solve(INode node, INode parent, int currentDepth) {
INode returnNode = null;
if (currentDepth > maxDepth) return returnNode;
//run rules
node = ruleEngine.run(node);
if (node.isGoalState()) {
   returnNode = node;
} else {
   if (node.isDeadEnd()) { // dead end
   //do nothing -- returns a null
  } else { //partial solution found...explore children
   while (true) {
         INode bestChild = node.getNextChild();
         if (bestChild != null) {
        returnNode = solve (bestChild, node, currentDepth + 1);
    }
         if (returnNode != null || bestChild == null) break; // continue until no more children or solution found
   }
  }
 }
 return returnNode;
 }
```

Putting it together

The main class that solves Sudoku in standalone mode is the SudokuSolver shown in the class diagram in Figure . It uses a WebSphere Operational Decision Manager implementation of the rule engine to execute the Sudoku business rules. The input and output to the rule engine is a SudokuWrapperNode, which is an implementation of the INode interface discussed in the previous section.

Figure 7. Sudoku Solver class diagram


«Java Class» SudokuSolver
MAX_DEPTH : int = 10;
engine : RuleEngine = new RuleEngine();
«use»
«Java Class» RuleEngine
rulesetPath : String = "sudoku-rules.jar";
run(INode) : INode
«Java Class» HeuristicDfsSolver
ruleEngine : IRuleEngine
maxDepth : int
HeuristicDfsSolver(IRuleEngine, int)
solve(INode, INode, int) : INode
«Java Class» SudokuWrapperNode
sudoku : Sudoku
guess : SudokuResolvedCellValue
isGoalState() : boolean
isDeadEnd() : boolean
registerSignificantEvent(ISignificantEvent) : void
getNextChild() : INode
applyBestGuess() : Sudoku
«Java Interface» IRuleEngine
run(INode) : INode
initializeWorkingMemory(INode) : void
reset() : void
«Java Interface» INode
isGoalState ()
isDeadEnd ()
registerSignificantEvent ()
getSignificantEvents ()
getNextChild ()

The rule engine uses an embedded WebSphere Operational Decision Management rule engine, or IlrContext. The rules for this engine are retrieved and parsed from a ruleset JAR during the initialization of the rule engine, as shown in the following snippet.

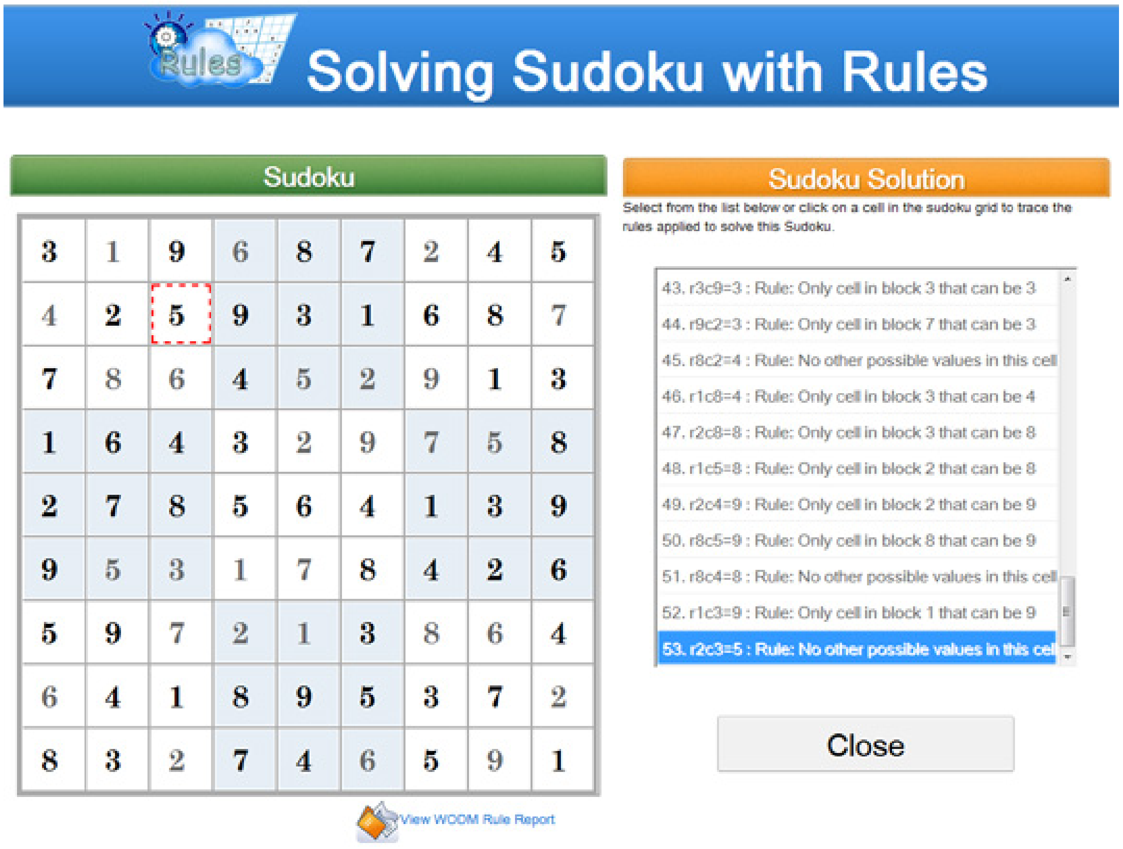
```
private void init() throws FileNotFoundException, IOException {
JarInputStream is = new JarInputStream(new FileInputStream(new File(
 rulesetPath)));

IlrRulesetArchiveLoader rulesetloader = new IlrJarArchiveLoader(is);
IlrRulesetArchiveParser rulesetparser = new IlrRulesetArchiveParser();

ruleset = new IlrRuleset();
rulesetparser.setRuleset(ruleset);

 boolean parsed = rulesetparser.parseArchive(rulesetloader);
 context = new IlrContext(ruleset);
 }
```

The SudokuNodeWrapper uses the most promising cell set by the rule engine to determine the next child to consider, as seen in line 15 in the SudokuNodeWrapper code snippet below. The next child is a clone of the current node to which the best guess on the most promising cell is applied.

```
1. @Override
2. public INode getNextChild() {
3. Sudoku childSudoku = applyBestGuess();
4. if (childSudoku == null) return null;
5. SudokuWrapperNode child =  new SudokuWrapperNode(childSudoku);
6. child.guess = guess;
7. return child;
8. }
9. /**
10.  * Clone current node and apply best guess
11.  * @return
12.  */
13. private Sudoku applyBestGuess() {
14. Sudoku child = null;
15. SudokuOpenCell openCell = sudoku.getMostPromisingCell();
16. if (openCell != null && openCell.hasCandidates()) {
17.  child = new Sudoku(sudoku);
18.  Collections.sort(openCell.getCandidates());
19.  int val = openCell.getCandidates().remove(0);
20.  guess = child.createResolvedCellValue(openCell.getRow(),
openCell.getCol(),
val, SudokuValueAssignmentType.GUESSED);
21. }
22. return child;
23. }
```

The SudokuSolver takes a string representation of the Sudoku grid and applies the hybrid approach discussed in this column. In my next column, we'll use a web application deployed to a cloud to display and solve Sudoku, as illustrated in Figure 8.

Figure 8. Sudoku solver on the web



3	1	9	6	8	7	2	4	5
4	2	5	9	3	1	6	8	7
7	8	6	4	5	2	9	1	3
1	6	4	3	2	9	7	5	8
2	7	8	5	6	4	1	3	9
9	5	3	1	7	8	4	2	6
5	9	7	2	1	3	8	6	4
6	4	1	8	9	5	3	7	2
8	3	2	7	4	6	5	9	1

Select from the list below or click on a cell in the sudoku grid to trace the rules applied to solve this Sudoku.

43. r3c9=3 : Rule: Only cell in block 3 that can be 3
44. r9c2=3 : Rule: Only cell in block 7 that can be 3
45. r8c2=4 : Rule: No other possible values in this cell
46. r1c8=4 : Rule: Only cell in block 3 that can be 4
47. r2c8=8 : Rule: Only cell in block 3 that can be 8
48. r1c5=8 : Rule: Only cell in block 2 that can be 8
49. r2c4=9 : Rule: Only cell in block 2 that can be 9
50. r8c5=9 : Rule: Only cell in block 8 that can be 9
51. r8c4=8 : Rule: No other possible values in this cell
52. r1c3=9 : Rule: Only cell in block 1 that can be 9
53. r2c3=5 : Rule: No other possible values in this cel

Conclusion

In this column, you've seen how rules can be used as part of a hybrid approach that also includes search with backtracking. In particular, I proposed an algorithm for the hybrid approach that you can use to solve constraint satisfaction problems, and applied it to solving Sudoku. This hybrid approach can solve any Sudoku puzzle, no matter how hard, as long as a solution exists.

You also saw that there is an inverse relationship between the amount of knowledge in the knowledge base and the need for trial and error. However, there is also an exponential relationship between the complexity of rules and the rule processing time. Therefore, you need to make a trade-off to determine at what point you should stop adding overly complex rules and resort to trial and error. You also saw how you can use rule task partitioning to mitigate some of the processing overhead of very complex rules.

Downloads

Description	Name	Size	Download method
Sudoku business rule report	Sudoku_Business_Rule_report.pdf	244KB	HTTP

Information about download methods

Resources

- Sudoku, Wikipedia
- Mathematics of Sudoku, Wikipedia
- The Sudoku Assistant
- Depth-first_search, Wikipedia
- IBM WebSphere Operational Decision Management V7.5 Information Center
- IBM ILOG ODM Developer Edition V3.6
- WebSphere Operational Decision Management V7.5 samples
- developerWorks BPM zone: Get the latest technical resources on IBM BPM solutions, including downloads, demos, articles, tutorials, events, webcasts, and more.
- IBM BPM Journal: Get the latest articles and columns on BPM solutions in this quarterly journal, also available in both Kindle and PDF versions.

About the author

Rajesh Rao

Rajesh (Raj) Rao has been working in the area of expert systems and business rule management systems for over 20 years, during which time he has applied business rules technology to build diagnostic, scheduling, qualification, and configuration applications across various domains such as manufacturing, transportation, and finance. He has been with IBM since2009. With a background in Artificial Intelligence, his interests include Natural Language Processing and Semantic Web.