Solving Sudoku Puzzles with Rewriting Rules [⋆]

Gustavo Santos-García [1]

Universidad de Salamanca

Miguel Palomino [2]

Dpto. de Sistemas Informáticos y Programación, Universidad Complutense

Abstract

The aim of the *sudoku* puzzle (also known as *number place* in the United States) is to enter a numeral from 1 through 9 in each cell of a grid, most frequently a 9×9 grid made up of 3×3 subgrids, starting with various numerals given in some of the cells (the "givens"). Each row, column, and region must contain only one instance of each numeral. In this paper we show how a sudoku puzzle can be solved with rewriting rules using Maude. Three processes (*scanning*, *marking up*, and *analysis*) are the classical techniques for solving sudokus. Elimination is the main strategy that we have employed. The strategy *what-if* and several contingencies are also implemented.

Key words: Rewriting logic, Maude, sudoku, puzzle.

1 Introduction

Rewriting logic [12] is a logic of concurrent change that can naturally deal with states and with highly nondeterministic concurrent computations. It has good properties as a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency. Moreover, it allows user-definable syntax with complete freedom to choose the operators and structural properties appropriate for each problem.

The naturality of rewriting logic and of its implementation, Maude [9], for modeling and experimenting with mathematical problems, has been illustrated in a number of works. Our goal with this paper is to further contribute to that pool but in a rather more recreational context, along the lines in [14].

[⋆] Work partially supported by Ministerio de Ciencia y Tecnología I+D+i Projects BFF2003-08998-C03-01 and TIC-2003-01000.

[1] Email:`santos@usal.es`

[2] Email:`miguelpt@sip.ucm.es`

For that, we present a case study of how to use Maude to execute and solve a popular kind of puzzles, namely sudokus. In Maude it is very easy to support objects and distributed object interactions in a completely declarative style with rewrite rules. We describe how sudokus can be represented in an object-oriented way and give formal rules that transform an initial sudoku into one in solved form, and how this representation can be straightforwardly mapped to Maude. The application of these rules, though always leading to a solution, can do so in many different ways; some give rise to combinatorial explosions and should thus be avoided. To handle this matter we have employed strategies to guide Maude's rewrite engine, which constitute an example of the use of the recently developed strategy language for Maude [13].

2 Sudokus

A (standard) **sudoku** is a 9×9 grid made up of 3×3 subgrids, also called "regions." Initially, some cells contain "given" numbers: the goal is to fill in the empty cells, one number in each, so that each column, row, and grid contains the numbers 1 through 9 exactly once (see an example in Figure 1). Originally

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

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

Fig. 1. A sudoku puzzle (©The Times, 2005, num. 295) and its solution.

called *number place*, the first such puzzle was created by Howard Garnes, a freelance puzzle constructor, in 1979 [1]. The puzzle was first published in New York by the specialist puzzle publisher Dell Magazines in its magazine *Math Puzzles and Logic Problems*. The puzzle was introduced in Japan by the publishing company Nikoli in the paper *Monthly Nikolist* in April 1984 as "*Suji wa dokushin ni kagiru*" [2], which can be translated as "the numbers must occur only once" or "*the numbers must be single*." At a later date, the name was abbreviated to *sudoku*, pronounced *sue-do-koo*; *su* = number, *doku* = single; it is a common practice in Japanese to take only the first kanji of compound words to form a shorter version. In 1986, Nikoli introduced two innovations which guaranteed the popularity of the puzzle: the number of givens was restricted to no more than 30 and puzzles became *symmetrical* (meaning the givens were distributed in rotationally symmetric cells). It is now published in mainstream Japanese periodicals, such as the *Asahi Shimbun*. The surge in popularity of sudoku in 2005 has led the world media to dub it as "the Rubik's cube of the 21st century" [3].

The numerals in sudoku puzzles are used for convenience. It works just as fine if the numbers are replaced with letters, shapes, colours or some other symbols. Sudoku is not a mathematical or arithmetical puzzle; arithmetic relationships between numerals are absolutely irrelevant. Indeed, Penny Press use letters in their version called *scramblets*; Knight Features Syndicate also use letters in their Sudoku Word (see Figure 2).

	R			N		B	A	
N				B		R		
		A	O					
			M					N
	S	N				E	M	
E					U			
					A	O		
		B		S				M
	E	O		M			S	

O	R	E	S	N	M	B	A	U
N	M	U	A	B	E	R	O	S
S	B	A	O	U	R	M	**N**	E
B	O	R	M	E	S	A	**U**	N
U	S	N	R	A	B	E	**M**	O
E	A	M	N	O	U	S	**B**	R
M	N	S	U	R	A	O	**E**	B
A	U	B	E	S	O	N	**R**	M
R	E	O	B	M	N	U	**S**	A

Fig. 2. A *Sudoku Word* puzzle (©Knight Features Syndicate, 2005) and its solution. Complete the grid so that every row, column and 3×3 box contains a different letter: A, B, E, M, N, O, R, S & U. One row or column contains a 7–letter word. What is it?

The Guardian calls these godoku and describes them as "devilish"; others name them *wordoku* or *sudoku word*. The required letters are given beneath the puzzle: once arranged they spell out a topical word between the top left and bottom right corners. This adds an extra dimension to sudokus as it may be possible to guess what the word is.

The appealing nature of the puzzle lies in the fact that the completion rules are simple, yet the reasoning required to reach the solution may be difficult. It also rises interesting questions, some of which are open: to measure the difficulty of a puzzle, to construct new sudokus, to establish the number of possible sudokus, to determine the number of possible puzzles with a single solution, to optimize the search of the solution and so on.

2.1 An NP Problem

Constraint programming has reached the masses. When solving their daily sudoku puzzle, thousands of newspaper readers apply classic propagation schemes in constraint programming like *X-wing* and *swordfish* [4]—patterns that cover several rows and columns, seeking a candidate number that can be removed from other lists in the corresponding columns and rows—to find a solution.

The general problem of solving sudoku puzzles on $n^2 \times n^2$ boards of $n \times n$ blocks is known to be NP-complete [15]. This gives some vague indication of why sudokus are hard to solve, but for boards of finite size the problem is also finite and can be solved by a deterministic finite automaton that knows the entire search tree. However, for a non-trivial starting board the search tree is very large and so this method is not feasible.

A valid sudoku solution is also a Latin square. A Latin square is an $n \times n$ table filled using different symbols in such a way that each symbol occurs exactly once in each row and exactly once in each column. Sudoku imposes the additional regional constraint; nonetheless, the number of valid sudoku solution grids for the standard 9×9 grid is $6,670,903,752,021,072,936,960$. This number is equal to $9! \times 72^2 \times 2^7 \times 27,704,267,971$, the last factor of which is prime. The result was derived through logic and brute force computation; the methodology for this analysis can be found at [10].

3 Definitions and Notation

3.1 Solution methods

As described in [11], three processes (*scanning*, *marking up*, and *analysis*) are the classical techniques for solving sudokus. The application of these processes is not deterministic so that the real performance of a sudoku solver depends on the approaches in their combination: if one applies these processes blindly, a combinatorial blowup may happen. Ideally, then, one needs to find a combination of these techniques which finds a solution in an efficient manner.

Scanning

It is performed at the outset and periodically throughout the procedure. It may have to be performed several times between analysis periods. Scanning consists of three basic methods that can be used alternatively: *cross-hatching*, *counting*, and "*looking for contingencies*."

Cross-hatching is the scanning of rows to identify which line in a particular grid may contain a certain number by a process of elimination. This process is then repeated with the columns. It is important to perform this process systematically, checking all of the digits 1–9. For fastest results, the numbers are considered in order of their frequency.

The *counting* of the occurrences of the numbers 1 to 9 in grids, rows, and columns tries to identify missing numbers. Counting based upon the last number discovered may speed up the search.

Advanced solvers look for "*contingencies*" while scanning, that is, narrowing a number's location within a row, column, or grid to two or three cells. When those cells all lie within the same row (or column) and grid, they can be used for elimination purposes during cross-hatching and counting. More difficult sudokus, by definition, cannot be solved only by basic scanning alone and require the detection of contingencies.

Marking up

Scanning comes to a halt when no more numbers can be discovered. At this point, it is useful to mark candidate numbers in the blank cells. Subscripts are a popular notation: the candidate numbers are written as subscripts in the cells. It is usually difficult to use this method in the newspaper because the cell space is small. There is a second notation with dots (a dot in the top left corner represents a 1 and a dot in the bottom right corner represents a 9). This notation has the advantage that it can be used on the original sudoku.

Analysis

There are two main analysis approaches: *elimination* and *what-if.* The *elimination* of possible numbers from a cell allows to leave the only possible choice. There are a number of elimination tactics. One of the most common is unmatched candidate deletion: a collection of n cells with identical possible numbers is said to be matched if the quantity of candidate numbers in each is equal to n. Then the numbers appearing as candidates elsewhere in the same row, column, or grid in unmatched cells can be deleted. For instance, if there are three cells in the same row with $\{2, 7, 8\}$ as their set of possible numbers, then 2, 7, and 8 can be discarded as possible numbers from the remaining cells in that row.

Using the *what-if* approach (which is called *reductio ad absurdum* in [11]), a cell with only two candidate numbers is selected and a guess is made. The procedure then continues with the resulting sudoku and, if no solution is found, the alternative number is tried.

3.2 Rules for solving sudokus

We will represent sudokus as a "soup" (formally, a set) of objects, where each object corresponds to a cell.

Definition 3.1 A **sudoku** S of order n can be represented as a set of objects $\mathcal{C}_{ij}$, $1 \leq i, j \leq n$, one for each cell at row i and column j, where each object has the following attributes:

- $G_{ij} = \sqrt{n} \cdot \mathrm{int}((i-1)/\sqrt{n}) + \mathrm{int}((j-1)/\sqrt{n}) + 1$ is the grid to which the cell belongs.
- P_{ij} is the set of possible numbers that may occur in this cell. The sudoku will be solved when this attribute is a unitary set for all cells; if P becomes empty for some cell, the sudoku does not have a solution.
- N_{ij} is the number of elements in P_{ij}.

Note that it is not necessary to consider i and j as additional attributes since they are part of a cell's name. The standard sudoku is a 9×9 grid, that is, a sudoku of order $n = 9$.

Initially P_{ij} is equal to $\{1, \ldots, n\}$ except when P_{ij} is the cell of a given g_{ij}, in which case $P_{ij} = \{g_{ij}\}$. Attributes G_{ij} and N_{ij} are stored explicitly but are computed from the rest of the elements of an object. We will apply several reduction rules to cells to obtain a solution for a sudoku. The goal is to reach a unique number in P_{ij} for all the cells; if P_{ij} becomes empty for some cell, then the sudoku has no solution.

During the procedure, and due to the what-if rule, a sudoku may be split into two and to distinguish one from the other we will enclose the corresponding objects in brackets. Thus, for example, the representation of two sudokus of order 9 will look like

$$[\mathcal{C}_{11}\mathcal{C}_{12}\ldots\mathcal{C}_{99}]\ [\overline{\mathcal{C}}_{11}\overline{\mathcal{C}}_{12}\ldots\overline{\mathcal{C}}_{99}].$$

Among the different processes (scanning, marking up, and analysis) for solving sudokus, the main strategy that we have considered is that of analysis for elimination. It is defined by Rule 1 below, which removes an element from the set of possible numbers in a cell. We have complemented it with Rules 2 and 3, of second and third order simplification, which respectively consider two and three elements. Several contingencies corresponding to the scanning process are covered by Rules 4 to 6. The strategy *what-if* is implemented with the sudoku split rules (Rules 7 and 8).

We next present in detail each rule. They are to be understood as local transition rules in a possibly concurrent system, that can concurrently be applied to different fragments of the soup: the cells in the upper part of each sequent become the ones below while the rest remain unchanged (since the attributes N_{ij} and G_{ij} are computed from P_{ij}, we do not explicitly mention their new values). We use $(\_)^c$ to denote the complement of a set.

Rule 1 (First order Simplification Rule) *If only one number is possible in a cell, then we remove this number from the set of possible numbers in all the other cells in the same row, column or grid. Symbolically,*

$$\frac{\mathcal{C}_{ij}\ \ \mathcal{C}_{i'j'}}{\mathcal{C}_{ij}\ \ \overline{\mathcal{C}}_{i'j'}}$$

where:

(i) $i = i'$ *or* $j = j'$ *or* $G_{ij} = G_{i'j'}$,

(ii) $P_{ij} = \{p\} \subseteq P_{i'j'}$,

and the attribute $\overline{P}_{i'j'}$ *of* $\overline{\mathcal{C}}_{i'j'}$ *is equal to* $P_{i'j'} - \{p\}$.

Rule 2 (Second order Simplification Rule) *If two cells in the same row (column or grid) have the same set of possible numbers and its cardinality is 2, then those numbers can be removed from the sets of possible numbers of every other cell in the same row (column or grid):*

$$\frac{\mathcal{C}_{ij}\ \ \mathcal{C}_{i'j'}\ \ \mathcal{C}_{i''j''}}{\mathcal{C}_{ij}\ \ \mathcal{C}_{i'j'}\ \ \overline{\mathcal{C}}_{i''j''}}$$

where

(i) $i = i' = i''$ *or* $j = j' = j''$ *or* $G_{ij} = G_{i'j'} = G_{i''j''}$,

(ii) $N_{ij} = N_{i'j'} = 2$,

(iii) $P_{ij} = P_{i'j'}$; $\quad P_{ij} \cap P_{i''j''} \neq \emptyset$, *and the attribute* $\overline{P}_{i''j''}$ *of* $\overline{\mathcal{C}}_{i''j''}$ *is equal to* $P_{i''j''} - P_{ij}$*.*

Rule 3 (Third order Simplification Rule) *If three cells in the same row (column or grid) have the same set of possible numbers and its cardinality is 3, then those numbers can be removed from the sets of possible numbers of every other cell in the same row (column or grid). Actually, the rule can be slightly generalized by allowing the cardinality of the sets to be 2 for some of the cells but one:*

$$\frac{\mathcal{C}_{ij}\ \ \mathcal{C}_{i'j'}\ \ \mathcal{C}_{i''j''}\ \ \mathcal{C}_{i'''j'''}}{\mathcal{C}_{ij}\ \ \mathcal{C}_{i'j'}\ \ \mathcal{C}_{i''j''}\ \ \overline{\mathcal{C}}_{i'''j'''}}$$

where

(i) $i = i' = i'' = i'''$ *or* $j = j' = j'' = j'''$ *or* $G_{ij} = G_{i'j'} = G_{i''j''} = G_{i'''j'''}$*,*

(ii) $N_{ij} = 3$; $\quad 2 \leq N_{i'j'}, N_{i''j''} \leq 3$,

(iii) $P_{i'j'}, P_{i''j''} \subseteq P_{ij}$*;* $\quad P_{ij} \cap P_{i'''j'''} \neq \emptyset$,

and the attribute $\overline{P}_{i'''j'''}$ *of* $\overline{\mathcal{C}}_{i'''j'''}$ *is equal to* $P_{i'''j'''} - P_{ij}$*.*

Further simplification rules could be added, but they are not needed and actually matching is much more expensive for them.

Rule 4 (Only One Number Rule) *When a number is not possible in any cell of a row (column or grid) but one, and the cardinality of the set of possible numbers for this cell is greater than one, then this set can become a singleton set containing that number. Symbolically,*

$$\frac{\mathcal{C}_{i_1j_1}\ \ \{\mathcal{C}_{i_kj_k}\}_{2\leq k\leq n}}{\overline{\mathcal{C}}_{i_1j_1}\ \ \{\mathcal{C}_{i_kj_k}\}_{2\leq k\leq n}}$$

where:

(i) *There exists a number* N*,* $1 \leq N \leq n$*, such that* $N = i_k$ *or* $N = j_k$ *or* $N = G_{i_kj_k}$ *for all* $1 \leq k \leq n$*,*

(ii) $N_{i_1j_1} > 1$,

(iii) *There exists a number* p*,* $1 \leq p \leq n$*, such that* $p \in P_{i_1j_1} \cap \left(\bigcup_{2\leq l\leq n} P_{i_lj_l}\right)^c$*,*

and the attribute $\overline{P}_{i_1j_1}$ *of* $\overline{\mathcal{C}}_{i_1j_1}$ *is equal to* $\{p\}$*.*

Rule 5 (Only Two Numbers Rule) *When two numbers* p_1 *and* p_2 *are not possible in any cell of a row (column or grid) but two, and the sets of possible numbers for these cells have cardinality greater than two, then these sets can become* $\{p_1, p_2\}$*.*

$$\frac{\mathcal{C}_{i_1j_1}\ \ \mathcal{C}_{i_2j_2}\ \ \{\mathcal{C}_{i_kj_k}\}_{3\leq k\leq n}}{\overline{\mathcal{C}}_{i_1j_1}\ \ \overline{\mathcal{C}}_{i_2j_2}\ \ \{\mathcal{C}_{i_kj_k}\}_{3\leq k\leq n}}$$

where:

(i) *There exists a number* N*,* $1 \leq N \leq n$*, such that* $N = i_k$ *or* $N = j_k$ *or* $N = G_{i_kj_k}$ *for all* $1 \leq k \leq n$*,*

(ii) $N_{i_1j_1}, N_{i_2j_2} > 2$,

(iii) *There exists two numbers p_1, p_2, $1 \leq p_1, p_2 \leq n$, such that $p_1, p_2 \in P_{i_1j_1} \cap P_{i_2j_2} \cap \left(\bigcup_{3\leq l\leq n} P_{i_lj_l}\right)^c$,*

and the attributes $\overline{P}_{i_1j_1}$ and $\overline{P}_{i_2j_2}$ of $\overline{\mathcal{C}}_{i_kj_k}$ are both equal to $\{p_1, p_2\}$.

Rule 6 (Twin Rule) *If, in a given grid, a number is only possible in one row (or column), then that number can be removed from the set of possible numbers in all the cells in that same row (or column) but different grid.*

$$\frac{\mathcal{C}_{i_0j_0} \quad \{\mathcal{C}_{i_kj_k}\}_{1\leq k\leq n}}{\overline{\mathcal{C}}_{i_0j_0} \quad \{\mathcal{C}_{i_kj_k}\}_{1\leq k\leq n}}$$

where:

(i) *There exists a number N, $1 \leq N \leq n$, such that $N = G_{i_kj_k}$, for all $1 \leq k \leq n$,*

(ii) *$i_0 \neq i_k$ and $j_0 \neq j_k$ for all $4 \leq k \leq n$,*

(iii) *$i_0 = i_1$ or $j_0 = j_1$,*

(iv) *There exists a number p, $1 \leq p \leq n$, such that $p \in P_{i_0j_0} \cap P_{i_1j_1} \cap \left(\bigcup_{4\leq l\leq n} P_{i_lj_l}\right)^c$,*

and the attribute $\overline{P}_{i_0j_0}$ of the representation of $\overline{\mathcal{C}}_{i_0j_0}$ is equal to $P_{i_0j_0} - \{p\}$.

Rule 7 (General Sudoku Split Rule) *This rule splits a sudoku when none of the other rules can be applied. We select a cell with a minimum number (greater than 1) of possible numbers. Then a sudoku is created with the first possible number and another one with the remaining possible numbers:*

$$\frac{\mathcal{C}_{ij} \quad \{\mathcal{C}_{kl}\}_{k\neq i, l\neq j}}{\left[\overline{\mathcal{C}}_{ij} \quad \{\mathcal{C}_{kl}\}_{k\neq i, l\neq j}\right] \quad \left[\overline{\overline{\mathcal{C}}}_{ij} \quad \{\mathcal{C}_{kl}\}_{k\neq i, l\neq j}\right]},$$

if these conditions are fulfilled:

(i) *$N_{ij} \geq 2$ and N_{ij} is minimal (but greater than 1),*

(ii) *$p \in P_{ij}$,*

and the attribute $\overline{P}_{ij}$ of $\overline{\mathcal{C}}_{ij}$ is equal to $\{p\}$ and the attribute $\overline{\overline{P}}_{ij}$ of $\overline{\overline{\mathcal{C}}}_{ij}$ is $P_{ij} - \{p\}$.

Rule 8 (Sudoku Split Rule) *This rule is the particular case of the previous one when the number of possible numbers in a cell is equal to 2.*

The correctness of each rule is immediate by their definition. Its application, however, does not lead to a confluent system: in case a sudoku has several solutions, the one that is reached can vary according to the order of execution of the rules; when a sudoku has a unique solution, this is obtained. The system is terminating: even though Rule 7 introduces a new sudoku, every rule decreases the cardinal of the set of possible numbers in one of the cells.

4 Rewriting logic and Maude

Maude [8,9] is a high performance language and system supporting both equational and rewriting logic computation for a wide range of applications. The key novelty of Maude is that besides efficiently supporting equational computation and algebraic specification it also supports rewriting logic computation.

A rewrite theory is a four-tuple $T = (\Omega, E, L, R)$, where (Ω, E) is a theory in an equational logic, L is a set of labels for the rules, and R is the set of labeled rewrite rules axiomatizing the local state transitions of the system. Some of the rules in R may be conditional. Mathematically, a rewrite rule has the form $l : t \longrightarrow t'$ **if** C, with t, t' terms of the same kind which may contain variables. Intuitively, a rule describes a local concurrent transition in a system: anywhere where a substitution instance $\sigma(t)$ of t is found, a local transition of that state fragment to the new local state $\sigma(t')$ can take place. We can regard an equational theory as the special case of a rewrite theory in which the sets of labels L and rules R are both empty; in this way, equational logic appears naturally as a sublanguage of rewriting logic.

Maude is a language whose modules are theories in rewriting logic. The most general Maude modules are called system modules and are written as `mod` T `endm`, with T the rewrite theory in question expressed with a syntax quite close to the corresponding mathematical notation. The equations E in the equational theory (Ω, E) underlying the rewrite theory $T = (\Omega, E, L, R)$ are presented as a union $E = A \cup E'$, with A a set of equational axioms introduced as attributes of certain operators in the signature Ω—for example, an operator $+$ can be declared associative and commutative with keywords `assoc` and `comm`—and where E' is a set of equations that are assumed to be Church-Rosser and terminating modulo the axioms A. Maude supports rewriting modulo different combinations of such equational attributes: operators can be declared associative, commutative, with identity, and idempotent.

4.1 Specifying sudokus in Maude

Our sudoku system module is called `SUDOKU`.

```
mod SUDOKU is
```

We first import into it some predefined modules that define the natural numbers, quoted identifiers, and string and number conversion. Maude provides useful support for modularity by allowing the definition of module hierarchies; a module can import other Maude modules as submodules in different modes, in this case with the keyword `protecting` (which can be abbreviated to `pr`):

```
  pr NAT .  pr QID .  pr CONVERSION .
```

The framework we have proposed for solving sudokus constitutes an example of an object-based system [9, Chapter 8]. Maude supports the specification of such systems in a simple and direct way through a predefined module called `CONFIGURATION`. However, instead of using this module, and since we want our output to be formatted in a tabular manner and to use different colours, we will explicitly declare all the operators needed for the specification of such systems; we will use them to introduce Maude syntax.

As described in the previous section, the cells in a sudoku correspond to objects and we will use messages to show the current status of the puzzle: `SearchingSolution`, `NoSolution`, `FinalSolution`. A "soup" of such objects and messages is called a configuration and will be used to represent sudokus. Thus, we declare sorts for objects and messages, which are subsorts of configurations, as well as for object and class identifiers (called `Oid` and `Cid`).

```
sorts Oid Cid Object Msg Configuration .
subsort Object Msg < Configuration .
```

To create objects we introduce, with the keyword `op`, an operator `<_:_|_>` that takes an object identifier, a class identifier, and a set of attributes as arguments.

```
op <_:_|_> : Oid Cid AttributeSet -> Object
   [ctor object format (n r! o g o tm! ot d)] .
```

The underbars allow the specification of mixfix syntax and are placeholders where the arguments should be written. (The commands enclosed in the brackets simply specify the format of the output [9].)

Configurations are declared with an operator with empty syntax (`__`) which is associative, commutative, and has an identity element.

```
op none : -> Configuration .
op __ : Configuration Configuration -> Configuration
        [ctor config assoc comm id: none] .
```

The current state of the solving procedure will be represented as a term of sort `Sudoku`, which consists of a set of sudokus each one enclosed in double angles:

```
sort Sudoku .
op none : ->  Sudoku .
op <<_>> : Configuration -> Sudoku .
op __ : Sudoku Sudoku -> Sudoku [ctor config assoc comm id: none] .
```

Attributes of objects belong to a new sort `Attribute`, which is a subsort of `AttributeSet` built with `_,_`.

```
sorts Attribute AttributeSet .
subsort Attribute < AttributeSet .
op none : -> AttributeSet .
op _,_ : AttributeSet AttributeSet -> AttributeSet
         [ctor assoc comm id: none  format (m! o sm! o)] .
```

For our sudoku specification we need to declare the three attributes associated to each cell: the grid to which it belongs, the set of possible numbers that may be placed in it, and its cardinality; the column and row will be given by the cell identifier. Attributes are declared as operators that return a term of sort `Attribute`.

```
op grd :_ : Nat -> Attribute .
op pss :_ : Set -> Attribute .
op num :_ : Nat -> Attribute .
op id : Nat Nat -> Oid .         --- row and column
```

All these declarations specify the core syntax needed to represent sudokus and our solving procedure in Maude. Now, their behaviour is specified by means of equations and rules. Equations are declared using the keyword `eq` and variables with the keyword `var`. For example, the operator that returns the grid associated to the cell at column `C` and row `R` can be specified as

```
op grd : Nat Nat -> Nat .
vars R C : Nat .
eq grd(R, C) = (sd(R, 1) quo 3) * 3  +  (sd(C, 1) quo 3)  +  1 .
```

where `sd` and `quo` the integer quotient are respectively the operators for symmetric difference and integer quotient.

An initial board for a sudoku can be specified as a constant term `sudoku`. To avoid the cumbersome task of explicitly writing the complete representation of a sudoku (9×9 cells with their corresponding rows, columns, numbers, grids, ..., for a sudoku of order 9), we use several auxiliar operators that will transform a much closer representation of a sudoku into the object-based format. For example, for the sudoku in Figure 1:

```
op sudoku : -> Sudoku .
eq sudoku =
   << msg('SearchingSolution)
      fill(1, 1,
      (  0 ; 0 ; 3 ;  7 ; 2 ; 0 ;  1 ; 0 ; 0 ;
         0 ; 0 ; 6 ;  9 ; 0 ; 5 ;  8 ; 0 ; 0 ;
         4 ; 9 ; 0 ;  0 ; 1 ; 0 ;  0 ; 5 ; 2 ;
         0 ; 5 ; 0 ;  0 ; 0 ; 0 ;  0 ; 6 ; 1 ;
         8 ; 0 ; 4 ;  0 ; 0 ; 0 ;  2 ; 0 ; 9 ;
         7 ; 6 ; 0 ;  0 ; 0 ; 0 ;  0 ; 4 ; 0 ;
         2 ; 3 ; 0 ;  0 ; 5 ; 0 ;  0 ; 1 ; 7 ;
         0 ; 0 ; 1 ;  2 ; 0 ; 3 ;  9 ; 0 ; 0 ;
         0 ; 0 ; 5 ;  0 ; 8 ; 6 ;  4 ; 0 ; 0 )
      ) >> .
```

Here, for example, the auxiliary operator `fill` places the givens in their corresponding cells, making thus their set of possible numbers a singleton, whereas the remaining cells will have $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ as their set of possible numbers. The sort `List` is an auxiliary sort over which lists of natural numbers are constructed using the operator _;_ in the usual way [9].

```
op fill : Nat Nat List -> Object .
eq fill(R, C, (N ; LL)) =
   if C == 9 then
      < id(R, C) : cell | grd : grd(R, C), pss : pss(N),
                          num : num(N) >
      fill(s R, 1, LL)
   else < id(R, C) : cell | grd : grd(R, C), pss : pss(N),
                            num : num(N) >
          fill(R, s C, LL)  fi .
eq fill(R, C, N) =
   < id(R, C) : cell | grd : grd(R, C), pss : pss(N),
                       num : num(N) > .
```

The operators together with the equations define the static part of the system. Next we need to define its dynamics, the way the system evolves towards reaching a solution, by means of rules that capture the processes for solving sudokus explained in Section 3.1. The goal, of course, is to find a solution whenever one exists (the set of possible numbers becomes a singleton for every cell) or otherwise to return a message warning about its non-existence (some set of possible numbers becomes empty).

Each of our rewrite rules diminishes the number of elements in the set of possible numbers of some cell, in a way that faithfully mimics the presentation of the solving rules given in Section 3.2. We illustrate the naturalness with which they are written in Maude by presenting two of them; for the rest, we refer the reader to the files at http://maude.sip.ucm.es/~miguelpt/bibliography.

The *Sudoku Split* rule that splits a sudoku into two when the number of possible numbers in a cell is equal to two, creating a sudoku with the first possible number and another with the second, is represented by means of

```
var VConf : Configuration .

rl [sudokuSplit2] :
   << msg('SearchingSolution)
      VConf < id(R, C) : cell | grd : G, pss : (P1 P2), num : 2 > >>
=> << msg('SearchingSolution)
      VConf < id(R, C) : cell | grd : G, pss : P1, num : 1 > >>
   << msg('SearchingSolution)
      VConf < id(R, C) : cell | grd : G, pss : P2, num : 1 > >>  .
```

This rule can be applied provided that a Sudoku term has a cell with just two elements P1 P2 in its set of possible numbers. In this case the term will be rewritten to a term with two concatenated Sudoku terms as expected: one with P1 as its set of possible numbers and another with P2.

As another example, and to illustrate the use of conditional rewriting rules, let us consider the *Second order Simplification* rule: if two cells in the same row (column or grid) have the same set of possible numbers and its cardinality is 2, then those numbers can be removed from the sets of possible numbers of every other cell in the same row (column or grid):

```
crl [simplify2nd] :
    < id(R1, C1) : cell | grd : G1, pss : (P P'), num : 2 >
    < id(R2, C2) : cell | grd : G2, pss : (P P'), num : 2 >
    < id(R3, C3) : cell | grd : G3, pss : (P LP3), num : N3 >
=>
    < id(R1, C1) : cell | grd : G1, pss : (P P'), num : 2 >
    < id(R2, C2) : cell | grd : G2, pss : (P P'), num : 2 >
    < id(R3, C3) : cell | grd : G3, pss :    LP3, num : sd(N3,1) >
    if ((R1 == R2) and (R1 == R3)) or ((C1 == C2) and (C1 == C3))
       or ((G1 == G2) and (G1 == G3)) .
```

The remaining rules for solving sudokus are represented in a similar manner.

In addition to the rules for solving a sudoku, there are also a number of equations and rules to take care of "maintenance" issues: to finish the procedure when there is no possible solution and to stop the application of rules when the final solution has been reached. For example, we have found a solution and therefore can stop the solving procedure if both the maximum and the minimum cardinality of the sets of possible numbers (computed by the auxiliary operators `maxCard` and `minCard`) are equal to 1:

```
ceq << msg('SearchingSolution) VConf >> =
    << msg('FinalSolution) VConf >>
    if (maxCard(VConf) == 1) and (minCard(VConf) == 1) .
```

Finally, to show the final term (a solved sudoku) in the desired format, a new attribute is added to the objects representing cells which is assigned the list of all the values in a given row:

```
  op val`:_ : List -> Attribute .
  eq msg('FinalSolution)
< id(R,1) : cell | pss: N1, At1 > < id(R,2) : cell | pss : N2, At2 >
< id(R,3) : cell | pss: N3, At3 > < id(R,4) : cell | pss : N4, At4 >
< id(R,5) : cell | pss: N5, At5 > < id(R,6) : cell | pss : N6, At6 >
< id(R,7) : cell | pss: N7, At7 > < id(R,8) : cell | pss : N8, At8 >
< id(R,9) : cell | pss: N9, At9 >
   = msg('FinalSolution)
< id(R,0) : rows | val: (N1 ; N2 ; N3 ; N4 ; N5 ; N6 ; N7 ; N8 ; N9) > .
```

Then, when a solution exists, the final term has the following form:

```
<< msg('FinalSolution)
  < id(1, 0) : rows | val : (2 ; 9 ; 5 ; 7 ; 4 ; 3 ; 8 ; 6 ; 1) >
  < id(2, 0) : rows | val : (4 ; 3 ; 1 ; 8 ; 6 ; 5 ; 9 ; 2 ; 7) >
  < id(3, 0) : rows | val : (8 ; 7 ; 6 ; 1 ; 9 ; 2 ; 5 ; 4 ; 3) >
  < id(4, 0) : rows | val : (3 ; 8 ; 7 ; 4 ; 5 ; 9 ; 2 ; 1 ; 6) >
```

```
    < id(5, 0) : rows | val : (6 ; 1 ; 2 ; 3 ; 8 ; 7 ; 4 ; 9 ; 5) >
    < id(6, 0) : rows | val : (5 ; 4 ; 9 ; 2 ; 1 ; 6 ; 7 ; 3 ; 8) >
    < id(7, 0) : rows | val : (7 ; 6 ; 3 ; 5 ; 2 ; 4 ; 1 ; 8 ; 9) >
    < id(8, 0) : rows | val : (9 ; 2 ; 8 ; 6 ; 7 ; 1 ; 3 ; 5 ; 4) >
    < id(9, 0) : rows | val : (1 ; 5 ; 4 ; 9 ; 3 ; 8 ; 6 ; 7 ; 2) > >>
```

4.2 Running the sudokus

The previous section has described how a procedure for solving sudokus can be specified in Maude. As discussed in Section 3.1, however, one cannot simply apply the rewrite rules in it to obtain a solution lest a combinatorial explosion is produced. Then, in order to avoid this it will be necessary to apply the rules according to some suitable strategy.

Maude has been extended with a powerful strategy language that allows a user to specify the ways in which the rewrite rules in a module will be applied [13]. This language is itself built on top of another extension, called Full Maude (see [9]) which adds to Maude a richer set of primitives for working with parameterization and object-oriented modules. We will not use any of the advanced features of Full Maude and for our purposes it will be enough to know that to load a module it is necessary to enclose it in parentheses. Regarding the strategy language, we will limit ourselves to a very limited subset; we refer the reader to [13] for the complete details.

A strategy module is declared with the keyword `stratdef`, strategy operators with `sop`, and the equations that define these operators are introduced with `seq`:

```
(stratdef STRAT is
  sop rules .
  seq rules = (simplify1st orelse
              (simplify2nd orelse
              (onlyOneNumber orelse
              (simplify3rd orelse
              (onlyTwoNumbers orelse
               twins))))) .
  sop split .
  seq split = (sudokuSplit2 orelse sudokuSplitN) .
  sop solve .
  seq solve = (rules orelse split) ! .
endsd)
```

This module defines three strategies. The first one, `rules`, simply tries to apply one of the first six rules, in order: it tries with `simplify1st`; if it is not possible, it tries with `simplify2nd`; and so on. The second one, `split`, tries to apply `sudokuSplit` and `sudokuSplitN` rules in order. The last one, `solve`, applies the first strategy and, only if it is not possible, tries to rewrite using splitting rules; the bang `!` at the end asks to continue with the application of the strategy while possible.

To illustrate its use, after loading Maude with Full-Maude and the module with the strategies we can solve the sudoku in Figure 1 by means of:

```
Maude> (srew sudoku using solve .)
rewrite with strategy :
result Sudoku :
  << msg('FinalSolution)
     < id(1,0): rows | val :(5 ; 8 ; 3 ; 7 ; 2 ; 4 ; 1 ; 9 ; 6) >
     < id(2,0): rows | val :(1 ; 2 ; 6 ; 9 ; 3 ; 5 ; 8 ; 7 ; 4) >
     < id(3,0): rows | val :(4 ; 9 ; 7 ; 6 ; 1 ; 8 ; 3 ; 5 ; 2) >
     < id(4,0): rows | val :(3 ; 5 ; 9 ; 8 ; 4 ; 2 ; 7 ; 6 ; 1) >
     < id(5,0): rows | val :(8 ; 1 ; 4 ; 5 ; 6 ; 7 ; 2 ; 3 ; 9) >
     < id(6,0): rows | val :(7 ; 6 ; 2 ; 3 ; 9 ; 1 ; 5 ; 4 ; 8) >
     < id(7,0): rows | val :(2 ; 3 ; 8 ; 4 ; 5 ; 9 ; 6 ; 1 ; 7) >
     < id(8,0): rows | val :(6 ; 4 ; 1 ; 2 ; 7 ; 3 ; 9 ; 8 ; 5) >
     < id(9,0): rows | val :(9 ; 7 ; 5 ; 1 ; 8 ; 6 ; 4 ; 2 ; 3) > >>
```

5 Final remarks

We have presented in this paper a case study of how to use Maude to execute and solve sudokus. We have first shown a representation of sudokus in an object-oriented way and introduced rules for solving them, whose implementation in Maude is straightforward. Since the blind application of these rules would give rise to a combinatorial explosion, we have explained how to take advantage of Maude's strategy language to apply the rules in a non-expensive way.

The main strength of our approach is the naturalness with which sudokus are represented in Maude and the ease with which the solving procedure is implemented. The specification can be easily modified to deal with sudokus of arbitrary order just by extending the rules with additional objects to represent the extra cells. However, it is precisely this need to add additional subterms to the rules which prevents the use of a single specification to cover all orders; such specification could be written by resorting to Maude's metalevel [9], but that would make the specification more obscure. To illustrate how the extension works, the `maude` files for the specification to solve sudoku monsters (sudokus of order 4), along with the complete specification of the sudoku solver presented in the paper, are available at http://maude.sip.ucm.es/~miguelpt.

On the other hand, the weakness of our implementation lies in its efficiency. Even with the use of strategies to prune the search tree, our implementation cannot compete with the numerous solvers available in the web (e.g. [5,6,7]).

Acknowledgements.

We thank Alberto Verdejo and Narciso Martí-Oliet for helpful comments and discussions (specially about the strategy language), and the anonymous

referees for their suggestions.

References

[1] History of Dell Magazines. http://www.dellmagazines.com/.

[2] *Rules and history from the Nikoli website.* http://www.nikoli.co.jp/puzzles/

[3] *The History of Sudoku*, Conceptis Editoria. http://www.conceptispuzzles.com/articles/sudoku/

[4] *Sudoku Solver*, Scanraid Ltd. http://www.scanraid.com/AdvanStrategies.htm

[5] SuDoku Solver by Logic v1.4. http://www.sudokusolver.co.uk/. (Javascript)

[6] Sudokulist. http://www.sudokulist.com/. (Step by step solver, with hints at each step.)

[7] A Su Doku solver. http://sudoku.sourceforge.net/. (Java)

[8] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer and José F. Quesada. Maude: Specification and Programming in Rewriting Logic, *Theoretical Computer Science*, 285(2):187–243, 2002.

[9] Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (version 2.2)*. http://maude.cs.uiuc.edu, 2006.

[10] Bertram Felgenhauer and Frazer Jarvis, *Enumerating possible Sudoku grids*, Univ. Sheffield and Univ. Dresden, 2005 http://www.shef.ac.uk/~pm1afj/sudoku/sudoku.pdf

[11] Michael Mepham. *Solving Sudoku*, Daily Telegraph (2005).

[12] José Meseguer, Conditional Rewriting Logic as a Unified Model of Concurrency, *Theoretical Computer Science*, 96(1):73–155, 1992.

[13] José Meseguer, Narciso Martí-Oliet, and Alberto Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004*, volume 117 of Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[14] Miguel Palomino, Narciso Martí-Oliet, and Alberto Verdejo. Playing with Maude. In Slim Abdennadher and Christophe Ringeissen, editors, *Fifth International Workshop on Rule-Based Programming, RULE 2004*, volume 124 of Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[15] Takayuki Yato and Takahiro Seta. Complexity and Completeness of Finding Another Solution and Its Application to Puzzles. *IEICE Trans. Fundamentals*, E86-A (5):1052–1060, 2003.