UDK: 004.02

Title: Spin for puzzles: Using Spin for solving the Japanese river puzzle and the Square-1 cube

Author(s): Evgeny V. Bodin (A.P. Ershov Institute of Informatics Systems),

Abstract: This paper describes how the SPIN model checker can be applied to solving puzzles, such as riverIQGame (an advanced "wolf, goat and cabbage" puzzle) or "Irregular IQ Cube" (also known as 'Square-1').

Keywords: SPIN, verification

1. Introduction.

Recently, formal verification of software and hardware systems becomes important for those systems whose reliabilty is crucial, for example, when the incorrectly used hardware may become dangerous to people (like medical equipment) or may be lost forever (like a spacecraft).

In the future, the need for verification specialists may increase, so it is important to prepare them now and to make young people aware of verification tools.

A possible way to popularize something is to solve puzzles with it.

The goal of the paper is to demonstrate how puzzles (non-trivial for a human being) can be solved by the SPIN verification tool [4]. Solving such puzzles can be useful for teaching students the modern automated verification tools based on the model-checking method.

Similar logical problems, like the famous "Farmer, Wolf, Goat and Cabbage problem"[1], are often used. But this problem is so well known that it becomes boring. Moreover, (almost) everyone already knows the answer. Another possible case study could be a search for fake coins among valid ones [5].

Some time ago, an advanced "Farmer, Wolf, Goat and Cabbage problem" puzzle appeared in the Internet[7], usually referred to as "Japanese river puzzle"[1]. The goal is to help eight people (a policeperson and a criminal, and a family consisting of a mother, a father and four children (two daughters and two sons)) cross the river using one small boat. The limiting rules are:

- the boat can hold at most two people (and at least one, since the empty boat cannot move);
- the father cannot stay with a daughter if the mother is not there;
- the mother cannot stay with a son if the father is not there;
- the criminal cannot stay with any person if the policeperson is not there;

[1]The strange thing is that the Flash game introduction is in Chinese.

- only the father, the mother and the policeperson know how to operate the boat.

A less known puzzle is the Irregular IQ Cube[11](see Fig. 1).[2]

In this paper, the above mentioned puzzles are solved using the SPIN model checker. SPIN (**S**imple **P**romela **In**terpreter) is a verification tool for parallel and distributed systems described in Promela (**Pro**cess/**Pro**tocol **Me**ta **La**nguage). As the SPIN website[9] says,

> Spin is a popular open-source software tool, used by thousands of people worldwide, that can be used for the formal verification of distributed software systems. The tool was developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments in the field. In April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM.

2. Japanese river puzzle specified in Promela.

First, use the Promela feature *mtype* for naming the moving entities in the puzzle:

```
mtype = {Cop, Criminal, Mom, Dad, Girl, Boy, Boat};
```

It assigns the values from 1 to 7 to these names.

Considering the original bank of the river as left, let the configuration be an array indexed with the entities, where the values represent the numbers of the corresponding entity at the right bank[3]:

```
int r[8];
```

It makes an array with indexes from 0..7, so the r[0] item is not used.

Obviously, the target condition is when the number of entities at the right bank is 1 or 2 for all of them:

```
#define DONE (r[Cop] == 1 && r[Criminal] == 1 && \
              r[Mom] == 1 && r[Dad] == 1 && \
              r[Girl] == 2 && r[Boy] == 2 && r[Boat] == 1)
```

The backslashes at the ends of lines allow a long line to be broken into several shorter ones.

Checking if a child is at the same bank with someone seems a little tricky, but it is easily seen from the following: e.g., when both boys are away from their mother, the values are:

[2]Strictly speaking, that thing is not a **cube** since not all its faces are squares.

[3]Choosing the right one is based on the fact that the Promela arrays are implicitly filled with zeroes at the beginning of the specification.

- either r[Mom] == 0 and r[Boy] = 2 (mother at left, both boys at right)
- or r[Mom] == 1 and r[Boy] = 0 (mother at right, both boys at left).

Using these definitions, let us describe the "unsafe" configuration:

```
#define NotWith(children, with) (r[children] == 2*(1-r[with]))
#define With(children, with)    (r[children] != 2*(1-r[with]))


#define CriminalUnsafe (r[Criminal] != r[Cop] && \
        (r[Criminal] == r[Mom] || r[Criminal] == r[Dad] || \
        With(Boy,Criminal) || With(Girl,Criminal) ))
#define BoysUnsafe ( With(Boy,Mom) && r[Mom]!=r[Dad] )
#define GirlsUnsafe ( With(Girl,Dad) && r[Mom]!=r[Dad] )
```

Since only three persons are allowed to drive the boat, let there be a *(driver, passenger)* pair (with a possible dummy passenger when the ‘driver’ goes alone). To avoid unnecessary duplication when the same pair is exchanged, let us order the possible drivers alphabetically: (Cop, Dad, Mom).

Here is the main part of the specification:

```
do
  /* move Cop (with anyone or alone) */
  :: r[Cop] == r[Boat] -> driver = Cop;
    /* Choose a 'random' passenger if it is here */
    if
        :: r[Criminal] == r[Boat] -> passenger = Criminal
        :: r[Mom] == r[Boat] -> passenger = Mom
        :: r[Dad] == r[Boat] -> passenger = Dad
        :: With(Boy,Cop) -> passenger = Boy
        :: With(Girl,Cop) -> passenger = Girl
        :: true -> passenger = 0 /* no passenger at all */
    fi;
    move(driver, passenger);
  /* move Dad (with a Boy or with Mom or alone) */
  :: r[Dad] == r[Boat]  -> driver = Dad;
    if
        :: r[Mom] == r[Boat] -> passenger = Mom
        :: With(Boy,Dad) -> passenger = Boy
```

```
        :: true -> passenger = 0
    fi;
    move(driver, passenger);
  /* move Mom (with a Girl or alone) */
  :: r[Mom] == r[Boat]  -> driver = Mom;
    if
        :: With(Girl,Mom) -> passenger = Girl
        :: true -> passenger = 0
    fi;
    move(driver, passenger);
   :: DONE -> printf("SOLVED\n"); assert(0); break;
   :: else -> printf("WHAT?!\n"); assert(0); break; /* Should never happen! */
od;
```

3. Applying SPIN to the Japanese river puzzle.

The resulting PROMELA specification[4] contains a 'do' loop whose parts (starting with '::') are executed non-deterministically (when several options are possible, they are chosen non-deterministically so that they are all searched when looking for a shortest path). When verifying, SPIN finds where the claimed assertions are violated. In this specification, there are two "assert(0);" statements that make SPIN 'think' it is an error. One of them (after the "DONE" condition) is really not an error, but the target. The other (marked with "else", which happens only when all other conditions are not met) would mean that the specification is wrong (or the puzzle cannot be solved). When one of them is reached, the verifier reports that fact and writes the 'trail', i.e. a file in a special format with a sequence of configurations that leads to the 'problem' configuration. When running with the '-t' option, SPIN uses this file to guide the execution of the specification, instead of choosing it non-deterministically.

There are two main methods for running Spin: command-line and GUI-based. The latter (using iSpin [10], jSpin or outdated XSpin) frees the user from the necessity to remember all command-line options. In this case, it is sufficient to make sure that the options "assertion violations" and "breadth-first search" are used.

Here is how the verification (using the command-line Spin) was made.

- `spin -a river_game1.pml`
  Make the source files for verification

- `gcc pan.c -DBFS -DREACH -DSAFETY -o file.pml.exe`

[4] Its complete text is given in the Appendix.

Compile the verifier[5].

- `file.pml.exe -E -n -i >__file.pml.exe.out_shortest__`
  Find a[6] shortest path to the final configuration.

- `spin -t river_game1.pml > __file1.pml.trailed__`
  Get a 'human-readable' representation of the path.

Results of running SPIN

The result of running `spin -t river_game1.pml` contains the moves that solve the river puzzle:

```
    Cop with Criminal go there.
    Cop goes back alone.
    Cop with Boy go there.
    Cop with Criminal go back.
    Dad with Boy go there.
    Dad goes back alone.
    Dad with Mom go there.
    Mom goes back alone.
    Cop with Criminal go there.
    Dad goes back alone.
    Dad with Mom go there.
    Mom goes back alone.
    Mom with Girl go there.
    Cop with Criminal go back.
    Cop with Girl go there.
    Cop goes back alone.
    Cop with Criminal go there.
    SOLVED
spin: river_game1.pml:108, Error: assertion violated
spin: text of failed assertion: assert(0)
spin: trail ends after 245 steps
#processes: 1
prev_dr = Cop
prev_pass = Criminal
```

[5] The `-DBFS` option makes the verifier use the breadth-first search method.
[6] There may be several shortest paths.

Figure 1: Irregular IQ Cube

```
r[0] = 0
r[1] = 1
r[2] = 2
r[3] = 2
r[4] = 1
r[5] = 1
r[6] = 1
r[7] = 1
245: proc  0 (:init:) river_game1.pml:111 (state 187) <valid end state>
1 process created
```

4. Irregular cube: a closer look.

After several rotations, the author failed to find his way back to the original cube configuration. Moreover, not only to its **original**, but even to **a cube**[7] configuration.

Of course, since this puzzle is not so widespread as the Rubik's cube, no complete instruction was found either. So, after some futile efforts I started to think of something that could help me.

I first met this puzzle at the think-geek site [11] where it was named 'Irregular IQ Cube', without any reference to its other names, so until recently I was not aware of the other names.[8]

[7]More or less, yes. Let us forget about it from now on and call it a cube anyway.

[8]Strangely enough, the above reference has not been valid for some time now, but it still can be found in

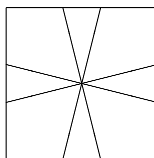

Figure 2: The target top face (the bottom face looks the same)

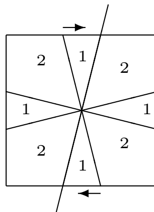

Figure 3: The target face, numbers and rotation

The first goal to reach was the “cube” form pretending that the colours are insignificant.[9]

4.1. Data representation.

The puzzle configuration can be represented as a pair of two faces: top and bottom. In the target configuration, each face is a square that consists of 8 parts: 4 triangles and 4 ‘deformed diamonds’ (or ‘kites’) (see Fig. 2). Let us denote the triangles and the kites by ’1’s and ’2’s, respectively (see Fig. 3). It has some meaning: the angle of a triangle is exactly half of the kite’s[10]. The sum of each face is $4 * 2 + 4 * 1 = 12$.[11]

Let us take a closer look: in order to be able to turn the halves of the cube, it must be possible to split each face by a straight line, so each face must be a pair of two halves, with the sum of the values of each half being 6. It makes sense not to consider those configurations of the faces that do not allow the cube’s halves to turn, since (a) such configuration is not a target configuration and (b) to reach a target configuration, a move to a ’good’ (where the ’halves-turn’ is possible) configuration is necessary.

Summarizing all the above, let us make a more formal description.

Configurations.

Cube configuration: a pair of faces.

Cube face: a pair of (good) half-faces.

Half-face: a list of several 1s and 2s with their sum being 6.

Target configuration: ((2, 1, 2, 1),(2, 1, 2, 1)), ((2, 1, 2, 1), (2, 1, 2, 1)) or (2121, 2121, 2121, 2121) for short.

Kaboodle [8].

[9]In fact, they have already become less distinguishable in the process.

[10]triangle’s angle is 30°, kite’s angle is 60°.

[11]That is, 12 times 30°.

Possible half-faces: 111111, 21111, 12111, 11211, 11121, 11112, 2211, 2121, 2112, 1221, 1212, 1122, 222.

Possible moves.

Face rotation. A circular clockwise shift of parts belonging to one face[12], so that the resulting face is also 'good'.

Halves rotation. Rotation of the halves so that one half of the top face is interchanged with one half of the bottom face.

Full cube rotation. The top face becomes the bottom face and vice versa[13].

5. Applying SPIN to the Irregular Cube.

To make things easier, a Perl script was written to solve a half of the problem. The script creates all possible moves that lead to a different good configuration[14].

The resulting PROMELA specification[15] consists of one process `solve()` with a lengthy 'do' loop whose parts (starting with '`::`') are executed non-deterministically (when several options are possible, they are chosen non-deterministically, so that they are all searched when looking for a shortest path). The 'MSC: ' part in the `printf` statements is used to make the messages appear in the 'Message Sequence Chart' that SPIN can create in the 'simulation mode'[16].

The verification (using the command-line Spin) was made similarly to the previous example.

- `perl make_tran1_1.pl >nul`
  Create the Promela file `file.pml`.

- `spin -a file1.pml`
  Make the source files for verification.

- `gcc -DBFS -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan.exe pan.c`
  Compile the verifier.

---

[12]The rotation is measured in 1s (the 30° angles), so that a half-face '222' cannot be rotated by an odd number.

[13]Let us rotate only the top face and use the full cube rotation if needed. It is the only reason to introduce this kind of rotation.

[14]So that moves like $(222, 222) \xrightarrow{Rotate\,2,\,4\,or\,6} (222, 222)$ are skipped because they lead to the same configuration, while $(222, 222) \xrightarrow{Rotate\,1,\,3\,or\,5}$ ... are simply invalid.

[15]A fragment of it is given in the Appendix.

[16]In this case, when executing the 'trail', the sequence of configurations leads to the target configuration, where the formula claimed to be true appeared false. In fact, the '`assert(0)`' is always false, so it is just a way to pretend that this location in the specification is unreachable to have verifier complain when it finds it reachable.

- `pan.exe -m10000  -E -n >__pan.exe.out__`

  Find a shortest path to the final configuration.

- `spin -t file1.pml >__file1.pml.trailed__`

  Get a ‘human-readable’ representation of the path.

- ***

  Handle the ‘real-life’ cube according to the path.

Human-readable trail path.

The following path is generated by running `spin -t file1.pml`.

```
      MSC: Initial state: m1221 m21111, m1221 m222
      MSC: Rotate 6 -> m21111 m1221, m1221 m222
      MSC: Rotate LEFT side -> m21111 m222, m1221 m1221
      MSC: Rotate 2 -> m2211 m1122, m1221 m1221
      MSC: Rotate LEFT side -> m2211 m1221, m1221 m1122
      MSC: Rotate 1 -> m1221 m1122, m1221 m1122
      MSC: Rotate 6 -> m1122 m1221, m1221 m1122
      MSC: Rotate LEFT side -> m1122 m1122, m1221 m1221
      MSC: Rotate 2 -> m2112 m2112, m1221 m1221
      MSC: Rotate LEFT side -> m2112 m1221, m1221 m2112
      MSC: Rotate 3 -> m2121 m1212, m1221 m2112
      MSC: Rotate LEFT side -> m2121 m2112, m1221 m1212
      MSC: Rotate RIGHT side -> m1221 m2112, m2121 m1212
      MSC: Rotate 3 -> m1212 m2121, m2121 m1212
      MSC: Rotate LEFT side -> m1212 m1212, m2121 m2121
      MSC: Rotate 2 -> m2121 m2121, m2121 m2121
      MSC: SOLVED
spin: file1.pml:1899, Error: assertion violated
spin: text of failed assertion: assert(0)
spin: trail ends after 26 steps
#processes: 1
t1 = m2121
t2 = m2121
b1 = m2121
b2 = m2121
 26: proc  0 (solve) file1.pml:1902 (state 2365) <valid end state>
1 process created
```

6. Conclusion.

This paper shows how to solve the "Japanese river puzzle" as well as the 'Irregular IQ Cube' [11].

In its preliminary version [3], the "topic of the 'future research" was specified as "to solve the cube completely", so that all cube faces have one color each. But all attempts to extend the approach 'failed miserably': verification of the Promela specifications generated by 'improved' Perl scripts (that took into account the colours of the parts) did not complete within a reasonable amount of resources (such as a 24-hour run on a machine with 12 GiB of RAM (all available RAM and swap space was consumed)). Possibly, this 'straight-forward approach' is a dead end.

At the same time, while trying to solve it, I discovered that the puzzle had a rather long history, as well as (more popular) alternative names with complete instructions (for example, at the Jaap Scherphuis' site [6][17]).

With those instructions, the puzzle can be solved without any computer. Some say that people with strong 'spacial thinking' can do it even 'intuitively', without any predefined rules.

There are people who also try to apply formal methods to puzzles or games like finding fake coins [5] or solving the checkers game [2].

References

1. Karpov, Yu. G.: *Model checking. Verification of parallel and distributed program systems*, BHV-Petersburg, 2010 (In Russian)
2. Baldamus, M., Schneider, K., Wenz, M. and Ziller, R.: Can American Checkers be Solved by Means of Symbolic Model Checking? *Electronic Notes in Theoretical Computer Science* Vol. 43, 2001, P. 15-33 Formal Methods Elsewhere (a Satellite Workshop of FORTE-PSTV-2000)
3. Bodin, E. V.: Puzzles and Spin: Irregular SPIN cube, *Program Understanding (Workshop at the 8th Ershov Informatics Conference)*, P. 4–9 Novosibirsk, 2011.
4. Holzmann, G. J.: *The Spin model checker — primer and reference manual*, Addison-Wesley, 2003.
5. Shilov, N. V. and Yi, K.: How to find a coin: propositional program logics made easy. *Current Trends in Theoretical Computer Science*, World Scientific. Vol. 2. 2004. P. 181-214.

[17] The 'Square-1' page also has a DOS/Windows program that solves an arbitrary cube using as much as 64 MiB of RAM

6. Jaap Scherphuis: (Back to) Square One / Cube 21 (online) - Jaap's Puzzle Page, URL: http://www.jaapsch.net/puzzles/square1.htm.
7. Japanese river puzzle as a Flash game, URL: http://freeweb.siol.net/danej/riverIQGame.swf.
8. Kaboodle site (online), URL: http://www.kaboodle.com/reviews/thinkgeek-irregular-iq-cube.
9. Spin model checker site (online), URL: http://spinroot.com/spin/whatispin.html.
10. Spin verifier's roadmap: Using iSpin, URL: http://spinroot.com/spin/Man/GettingStarted.html.
11. ThinkGeek site (online), URL: http://www.thinkgeek.com/geektoys/games/9766/.
12. Wikipedia page on Square-1 cube (online), URL: http://en.wikipedia.org/wiki/Square_One_(puzzle).

7. Appendix 1. The resulting PROMELA specification for the Cube.

```
/* Cube Solve */

mtype= {m111111, m21111, m12111, m11211, m11121, m11112,
        m2211, m2121, m2112, m1221, m1212, m1122, m222 };
#define T1 m1221
#define T2 m21111
#define B1 m1221
#define B2 m222
#define FINAL m2121


/* Initial configuration */
mtype t1 =T1, t2=T2, b1 =B1, b2 =B2;

active proctype solve() {
  mtype tmp =0;

  printf("MSC: Initial state: %e %e, %e %e\n", t1, t2, b1, b2 );


  do
  /* Rotate */
  :: t1==m111111 && t2==m2211 -> d_step{
     t1=m21111 ; t2=m11112;
     printf("MSC: Rotate 4 -> %e %e, %e %e\n", t1, t2, b1, b2);
```

```
    }
  :: t1==m111111 && t2==m2211 -> d_step{
    t1=m2211 ; t2=m111111;
    printf("MSC: Rotate 6 -> %e %e, %e %e\n", t1, t2, b1, b2);
    }
```

. . . (Many, many similar statements)

```
  :: t1==m222 && t2==m1122 -> d_step{
    t1=m1122 ; t2=m222;
   printf("MSC: Rotate 6 -> %e %e, %e %e\n", t1, t2, b1, b2);
    }

  /* Switch halves */
  :: /* true -> */ d_step{
       tmp = t2; t2 = b2; b2 = tmp; tmp=0;
   printf("MSC: Rotate LEFT side -> %e %e, %e %e\n", t1, t2, b1, b2);
       }
  :: /* true -> */ d_step{
       tmp = t1; t1 = b1; b1 = tmp; tmp=0;
   printf("MSC: Rotate RIGHT side -> %e %e, %e %e\n", t1, t2, b1, b2);
       }
  /* Full Cube Move */
  :: true -> d_step{
       tmp = t1; t1 = b1; b1 = tmp;
       tmp = t2; t2 = b2; b2 = tmp;
       tmp=0;
   printf("MSC: Full Cube Move -> %e %e, %e %e\n", t1, t2, b1, b2);
       }
  /* Pretend that the final state is not reachable
    to have SPIN find a counter-example  */
  :: t1==FINAL && t2==FINAL && b1==FINAL && b2==FINAL ->
       printf("MSC: SOLVED\n"); assert(0); break;
  :: else -> printf("MSC: WHAT?");
       assert(0); break; /* Should never happen! */
  od;
} /* solve() */
```

8. Appendix 2. The complete PROMELA specification for the Japanese river puzzle.

```
/* Japanese WolfGoatCabbage Puzzle Solve */
mtype = {Cop, Criminal, Mom, Dad, Girl, Boy, Boat};

#define DONE (r[Cop] == 1 && r[Criminal] == 1 && \
              r[Mom] == 1 && r[Dad] == 1 && \
              r[Girl] == 2 && r[Boy] == 2 && r[Boat] == 1)

#define NotWith(children, with) (r[children] == 2*(1-r[with]))
#define With(children, with)    (r[children] != 2*(1-r[with]))

#define CriminalUnsafe (r[Criminal] != r[Cop] && \
        (r[Criminal] == r[Mom] || r[Criminal] == r[Dad] || \
        With(Boy,Criminal) || With(Girl,Criminal) ))
#define BoysUnsafe ( With(Boy,Mom) && r[Mom]!=r[Dad] )
#define GirlsUnsafe ( With(Girl,Dad) && r[Mom]!=r[Dad] )



mtype prev_dr   = 0;
mtype prev_pass = 0;

inline printMove(driver, passenger, boat)
{
    if
        :: boat == 0 ->
            if
            :: passenger == 0 ->
                printf("%e goes there alone.\n", driver);
            :: else  ->
                printf("%e with %e go there.\n", driver, passenger);
            fi;
        :: else ->
            if
            :: passenger == 0 ->
                printf("%e goes back alone.\n", driver);
            :: else  ->
                printf("%e with %e go back.\n", driver, passenger);
            fi;
    fi;
```

```
}

inline update_r()
{
  r[driver] = r[driver] + (1-2*r[Boat]);
  if
     :: passenger != 0 -> r[passenger] = r[passenger] + (1-2*r[Boat]);
     :: else -> skip;
  fi;
  r[Boat] = 1 - r[Boat];
}

inline move(dr, pass)
{
  printMove(dr, pass, r[Boat]);
  if
    :: (dr == prev_dr && pass == prev_pass) -> printf("Don't do the same move!\n");
    :: else ->
        update_r();
        if
        :: (CriminalUnsafe || BoysUnsafe || GirlsUnsafe) ->
            /* undo move */
            update_r();
         :: else ->
            prev_dr = dr; prev_pass = pass;
        fi;
  fi;
}


/* Global array for positions, initially = 0 */
int r[8];
/* mtypes are assigned from 1, array are indexed from 0, so the r[0] is not used */

init {
    local mtype driver    = 0;
    local mtype passenger = 0;
```

```
    /* Run */
    do
      /* move Cop (with anyone or alone) */
      :: r[Cop] == r[Boat] -> driver = Cop;
        /* Choose a 'random' passenger if it is here */
        if
            :: r[Criminal] == r[Boat] -> passenger = Criminal
            :: r[Mom] == r[Boat] -> passenger = Mom
            :: r[Dad] == r[Boat] -> passenger = Dad
            :: With(Boy,Cop) -> passenger = Boy
            :: With(Girl,Cop) -> passenger = Girl
            :: true -> passenger = 0 /* no passenger at all */
        fi;
        move(driver, passenger);

      /* move Dad (with a Boy or with Mom or alone) */
      :: r[Dad] == r[Boat]  -> driver = Dad;
        if
            :: r[Mom] == r[Boat] -> passenger = Mom
            :: With(Boy,Dad) -> passenger = Boy
            :: true -> passenger = 0
        fi;
        move(driver, passenger);

      /* move Mom (with a Girl or alone) */
      :: r[Mom] == r[Boat]  -> driver = Mom;
        if
            :: With(Girl,Mom) -> passenger = Girl
            :: true -> passenger = 0
        fi;
        move(driver, passenger);

       :: DONE -> printf("SOLVED\n"); assert(0); break;
       :: else -> printf("WHAT?!\n"); assert(0); break; /* Should never happen! */
    od;
}
```

УДК: 004.02

Название: Spin для головоломок: Использование системы проверки моделей SPIN

для решения японской головоломки о переправе через реку и головоломки "Куб-1".

Автор(ы): Бодин Е.В. (Институт систем информатики СО РАН)

Аннотация: Статья описывает применение системы проверки моделей SPIN к решению японской головоломки о переправе через реку (продвинутый вариант задачи о волке, козе и капусте) и головоломки "Irregular IQ Cube" (также называемой "Куб-1").

Ключевые слова: SPIN, верификация