Applying Formal Techniques
to Solving Rubik's Cubes

SatRubiks

Nathan Scheirer, Joshua Wallin, Patrick Sivets

December 11, 2017

1   Introduction

At some point in life the question has had to have come up; how can I solve a Rubik's cube? For some of us that takes a lot of practice and studying of various algorithms to solve any scrambled cube that's given to us. However, for the rest of us, who don't have time to learn all these different algorithms, yet still want the satisfaction of solving the most popular toy ever created there must be an easier way. That's where the concept of solving a Rubik's cube with formal methods comes into play. How nice would it be to be able to just input the starting condition of the cube into your solver and have a sequence of moves spit back out that exactly solves your puzzle? In our opinion, that would be very nice. So in this study we take a look at a few various formal method solvers to see if this application is feasible and, if so, which is the best method to use.

2   Spin Model Checker

2.1   Background

The SPIN Model Checker was developed in the early 1980s at Bell Labs and has been freely available for use since 1991. SPIN is a popular verification tool using the Promela (Protocol Meta Language) language. However, Spin has not been used in conjuncture with solving puzzles. Upon researching Spin it was only seen in one piece of literature where Spin was used as a puzzle solver. In the paper *Spin for puzzles: Using Spin for solving the Japanese river puzzle and the Square-1 cube* by Evgeny V. Bodin et al., they use Spin to model and solve common puzzles with the use of planning. Spin can be used in a very similar manner to solve the 2x2 Rubik's Cube.

2.2   Set-Up

2.2.1   Initial Programs

Correctly representing the Rubik's cube and its transitions in Promela proved to be a tricky task. Our initial attempts to create a working model lead to a long list of errors and rather inefficient code. The initial model had several places in the code where errors were present. First, there was an inconsistency with our description of the cube in Promela. The physical cube did not reflect what was encoded into our model. This oversight led to further errors including, misrepresentation of transitions and incorrect final states. To correct these fatal errors, a new model needed to be developed to accurately depict the 2x2 Rubik's cube.

In order to achieve an accurate representation, a simple model was created to represent a "two-sided" cube. This is an idea where a cube only has two sides and where those sides are each a different color and the model must accurately

swap these colors during a transition. This "two-sided" cube was modeled using the code:

```
byte a = 1, b = 0;

byte temp1 = 0;

#define FINISH ((a == 0) && (b == 1))

active proctype test()
{
    do
    ::  assert(!FINISH) -> atomic {
         temp1 = a;
         a = b;
         b = temp1;
       }
    od
}
```

In this code each side is represented using bytes a and b, which are both assigned a color using boolean values. To define the transition, a variable is defined to act as a temporary value store. Within the do-loop is where the transition occurs and by defining the transitions in this way, the model was able to accurately able to give a counter-example leading to the final state as defined in the code.

The idea from this simple "two-sided" cube was extended to a normal six-sided cube with each face a different color. This model was very similar in how it was written, using variable to temporarily store the sides color information during the transitions. One main difference in this code is that there needed to be two different transitions represented, one in the vertical direction and one in the horizontal. To do this it was defined that the horizontal direction would rotate clockwise and the vertical direction upwards. The code for the horizontal transition is shown below.

```
byte a=1, b=2, c=3, d=4, e=5, f=6;
byte tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, tmp8;
#define FINA (a==4)
#define FINB (b==1)
#define FINC (c==2)
#define FIND (d==3)
#define FINE (e==5)
#define FINF (f==6)
#define FINAL (FINA && FINB && FINC && FIND && FINE && FINF)
active proctype test(){
    do
```

```
:: assert(!FINAL)-> atomic{
printf("right");
tmp1=a;
tmp2=f;
tmp3=c;
tmp4=e;
f=tmp1;
c=tmp2;
e=tmp3;
a=tmp4
}
```
.
.
.

An issue that this code brought to light is how to appropriately define the final state in Promela. As shown above that was done by using the define function and setting the final state of each side. This removed the need for long LTL properties and it kept the code clean and concise. To verify that this model represented the six-sided cube appropriately it had to be checked that the final state could actually exist. When running the code with an end state that did not exist the code would not give a trace stating how we could reach that state.

2.2.2   2x2 Rubik's Cube

The same ideas that were used in the simple initial programs could be applied to the more complex 2x2 Rubik's Cube. However, in order to accurately represent the cube and its movements, the cube must be appropriately represented in Promela. To do this a system was implemented that labeled each "face-let" of the cube in accordance to its orientation. This is shown in the picture below.
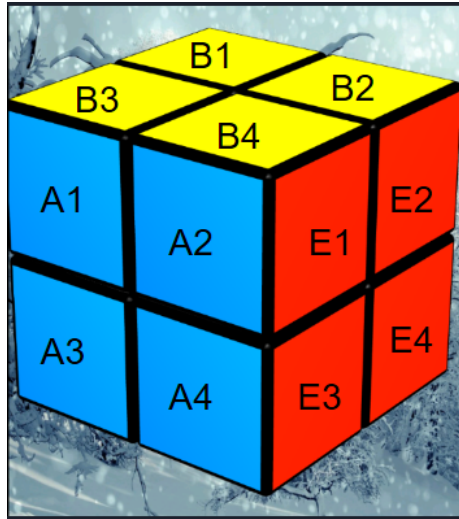
As shown in the picture, the 2x2 cube's face-lets were defined with each face receiving a designated letter (A-F) and that letter was paired with a number corresponding to the face-let position, with the number one beginning in the top left through four in the bottom left corner. It was also encoded to represent each color by an integer value. Labeling the cube in this consistent way allowed for the transitions to be reflected accurately.

To accurately reflect these transitions it must be known that there are only six different moves that can be made with a standard 2x2 Rubik's Cube.
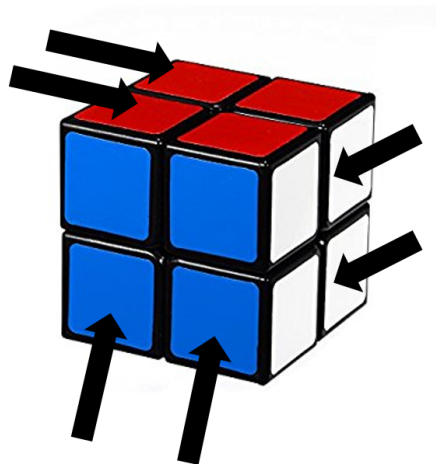
These moves, reflected above, were all set to rotate in a clockwise direction. Again, this consistency allowed for the encoded transitions to reflect the physical cube's movements.

The final aspect of the cube that needed to be encoded was the final state that the program was attempting to reach. Initially, this proved to be a very messy and long LTL property due to the fact that each side could have six possible colors when the cube was solved. However, the SatRubiks team found that this issue could be cleaned up by working backwards. Meaning that instead of having an initial state be the random configuration of the cube, the initial state could be encoded to represent a solved cube and the final state would then represent the random configuration. This idea removed the need for nasty LTL properties and made made the code relatively easy to read and understand.

Now that the cube has been setup accurately with Promela and the transitions have been correctly encoded, the ideas that were utilized in the initial programs are ready to be applied to the 2x2 Rubik's cube. The full Promela code for the 2x2 Rubik's cube can be found in Appendix B or on GitHub in the SatRubiks repository.

It is important to note that when running this code with Spin some of the settings will have to be adjusted. For instance, when running this code the "physical memory available" will have to be adjusted from its default setting to that of at least 1,000,000 MB. If this setting is not adjusted it causes catastrophic errors within Spin and the program will fail to run. Another important aspect of running this code is that the search mode will need to be changed from depth-first search to breadth-first search. This will change the way that Spin searches the state-space of the cube. Instead of "walking through" each node one by one as it does in a depth search, Spin will search the node's nearest neighbors before moving on down the state-space tree. This significantly reduces run times. When run with the depth-first search set the program was seeing run times anywhere from 30 minutes to well over an hour. Changing this setting cut the run times down to the level of seconds to a few minutes.

2.3 Results

In order to generate random configurations of a 2x2 Rubik's cube, the SatRubiks team used a popular application for the iPhone, *Cube 3D Kit*. This application provided a platform that could not only generate random initial conditions but also allow the user to attempt to solve the puzzle. Using this application, several random configurations were generated and ran against the Spin model. Depending on the initial configuration of the cube the model could solve it in as little as half a second, however in some uncommon cases it could take up to a couple minutes. Below are two different configurations of the 2x2 cube that were checked against the model.

By inputting the appropriate orientation into the Spin model, the algorithm
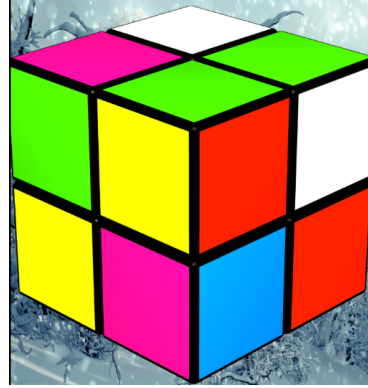
Figure 1: Random Config.A

Figure 2: Random Config. B

provided a correct path that led to the solved cube when followed in reverse. Note that each clockwise transition will now represent a counter-clockwise transition due the backward nature of the program. In the first case, the algorithm was able to solve the puzzle in just .49 seconds. Provided a path that consisted of seven moves to be solved. The second configuration took a little longer for the algorithm to solve, solving it in 10.9 seconds. This amount of time was rather common when running other tests. Only in a few instances was there long run times of several minutes. However, there was not a lot of consistency in the run times and some instances where this model was not able to solve the initial configuration given.

2.4   Verification and Future Work

Attempting to make sure that the model accurately reflected the cube and its moves was one of the more difficult tasks. There was no real good way to validate the transitions aside form the use of inspection. To make this as accurate as possible, all transitions were explicitly written out including what each face-let was doing during a single rotation. These were then compared with the encoded information to ensure an that the physical cube was appropriately reflected.

The model was also verified by using the way the cube is constructed and checking to see if certain configurations are allowed in the model. For instance, in the cube above yellow and white are on opposite sides of the cube and thus can never be next to each other in any configuration. By using simple LTL propositions, such as !(b3 == 2 && a1 == 4) if two and four represent yellow and white receptively, this can be check to be sure that the model does not allow this to occur. This same principle can also be applied to trying to find paths that lead to impossible end states in our model. Running the model using a final state that represents white and yellow being next to each other the algorithm should not produce any trace that allows for this either.

7

There is still more work needing to be done with the 2x2 model. The model should be able to solve every configuration that it is given. This could be a memory issue within Spin, it is not clear. However, expanding this model to the 3x3 cube should not be overly difficult considering how the 2x2 cube is represented and how the transitions are encoded. The concern with this is with the memory limitations that were encountered using the Spin for the 2x2 model.

3 Pure SAT Solving

3.1 Background

Satisfiability solving (SAT) is a method for determining the satisfying solution to a given boolean equation. SAT solving was the first problem to be determined NP-complete via the Cook-Levin theorem. Therefore, this formal method provides a unique and promising solution to the problem discussed in this paper, the Rubik's cube.

3.2 Set-Up

3.2.1 Method

In regards to solving a Rubik's cube with formal methods, using a Satisfiability Solver (SAT) has been discussed the most throughout literature [1]. Therefore, this approach, initially, appeared to be the most straightforward of the processes discussed in this paper. This proved to be true as work started on developing the pure SAT model. With the aid of a program called Sabr [2], creating SAT queries for the Rubik's cube became not only trivial but extremely simple. Not only were the models easy to develop but Sabr also parses the output of any given SAT solver into a format that is easy to read and follow. This allowed us to compare the efficiency of the Sabr parsing with that of NuXmv as well as the efficiency of a variety of SAT solving back ends. The overall process for solving a Rubik's cube via SAT can be observed in figure 3:
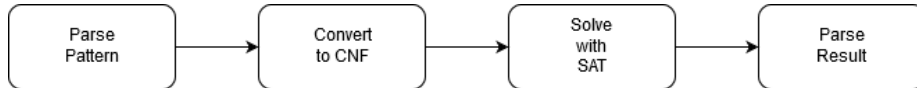

Figure 3. SAT Solving process

3.2.2 Parsing with Sabr

This input to Sabr comprises of a few different main definitions: Board, Start, End, and DesObjs. With these components a model can be created as seen in model Appendix B. The initial state of an unsolved Rubik's cube is placed in the Start definition with the following structure:

```
         y w y
         w   g
         o y r
o r g    w b g    y y b    r b g
o   o    b   w    g   y    r   w
r g b    w b o    y o r    w o w
         o r b
         r   g
         g y b
```

Figure 4. Initial Sabr model condition

Sabr then converts this initial condition, along with the transition definitions, into a SAT query that can be inputted into any SAT solver that utilizes the conjunctive normal form (CNF) formula structure. CNF is a form of logical equation that is constructed of ands of clauses of ors. An example of a CNF formula can be observed below where each new line is denoted with a 0 and each line is a new clause. Each variable can either be positive or negative, where negative numbers represent the negation of that variable. The first line of a CNF formula (starting with a "p") expresses the number of variables and the number of clauses in the generated formula. These numbers were useful later in determining the correctness of the developed models because the number of variables used to solve a Rubik's cube is known.

p cnf 2484 23442 0 71 0 113 0 211 0 295 0 337 0   Equation 1

3.2.3   SAT Solvers

Two SAT solvers were considered during this project, the first being a SAT solver created during the 2017 SAT competition called CaDiCaL [3] and the second being the popular solver miniSat. MiniSat was chosen because of its popularity and it is also the back-end for NuXmv which allowed for another avenue of comparison between the proposed models. CaDiCaL was chosen for its performance in the agile track during the 2017 SAT solver competition. Placing first in this category showed that this was one of the quickest and simplest solvers created just recently. This created an interesting dimension to the comparison described throughout this paper.

3.3   Results

Initially, a 2x2 model of the Rubik's cube was created to determine the feasibility of the proposed SAT method. What soon became obvious was that this method was very easy to work with and the results were proving to be accurate after a few test runs of the model. Only a few patterns that were tested resulted in long run times and were unable to be solved. Following the success of the 2x2

9

model a 3x3 model was created and analysis was performed on the results of that model as well. Unlike the 2x2 model, the 3x3 model proved to be much more difficult to get results, with most patterns running for days without a solution. This made it difficult to produce usable results from the 3x3 model, however the method was verified with the 2x2 model and the results from that are still very comparable to the other methods described in this paper.

3.3.1 2x2 Model

Average benchmarking results can be observed in table 1 below with extended results available in the Github source and in A1.

Table 1. 2x2 Rubik's Cube average results

2x2 Rubik's Cube		
	CPU Time (s)	Memory Used (Mb)
Sabr/miniSAT	216.4	31.41
CaDiCaL	43.96*	12.394

* Encountered runtimes > 200,000 seconds

3.3.2 3x3 Model

Average benchmarking results can be observed in table 2 below with extended results available in the Github source and in A2.

Table 2. 3x3 Rubik's Cube average results

3x3 Rubik's Cube		
	CPU Time (s)	Memory Used (Mb)
Sabr/miniSAT	5041.31	305.36
CaDiCaL	NA	NA

3.3.3 Verification

The models discussed here were verified using a few different techniques. Among those are visual inspection, random testing, and intentional error introduction. Due to Sabr's simple layout and requirements determining the correctness of the input models was a non-issue just by visual inspection. This was further supported by receiving correct counterexamples that led to a valid cube solution. A variety of different patterns were also tested to guarantee that the model was producing accurate results. This was also validated by the use of different SAT solvers which allowed for the comparison of their respective outputs. Errors were also intentionally placed within the model to validate that the resulting CNF formula was not satisfiable. This was done by placing an incorrect amount of a certain color of face into the model and by putting colors that cannot physically exist next to each other. After this verification, the accuracy of the results of these models can be considered reasonably high.

3.3.4 Conclusions

As clear as the results shown in table 1 appear to be, the extended results shown in A1 and A2 express a different yet complicated set of data. Given the average data, one would say that CaDiCaL is the clear winner in terms of memory usage and runtime, however when looking at the efficiency of the solver itself, in terms of number of propagations, decisions, and amount minimized, the winner looks much more like miniSat. Therefore, miniSat could be considered more robust than CaDiCaL in that it solves more patterns in about the same amount of time, compared to CaDiCaL, which could solve some patterns very quickly, but others in an unreasonable amount of time. Despite the limited amount of data for the 3x3 cube this conclusion could be extrapolated to the increased level of computation. The advantage of CaDiCaL, however, is that it uses much less memory than miniSat which for systems with little memory, then CaDiCaL may be the better option. Overall, using Sabr with it's miniSat back-end is a very easy to use, accurate, and relatively efficient method for solving a Rubik's cube.

3.3.5 Future Work

More work needs to be done in solving the issues with the SAT solvers not being able to solve some of the patterns in a reasonable amount of time. Even the patterns that miniSat could solve, yet CaDiCaL failed, shows that there is something in CaDiCaL that may have been over looked. Unfortunately, there will always be memory issues when dealing with this immense amount of variables, but future work in making that number even smaller will only allow for a decrease in runtime and memory usage.

4 A Bounded Model Checking Approach

4.1 Background

When model checking an incredibly large finite (or potentially infinite) state space, a natural approach consists of limiting the search to only those states that are deemed "relevant"; in many cases, this "relevance" may be defined by a proximity to the start of the model, a finite limit on the distance from the initial state that is to be checked. The number of steps from the initial state, k, that must be visited to demonstrate adherence of a system to a particular property is referred to as the *completeness threshold* [3]. For a property, such as $\Box\varphi$, k is selected such that all states are reachable within k steps from the initial state. Thus, showing that no property violation occurs in this bound is sufficient to demonstrate that no violation occurs for any valid run of the system.

4.2 Method

The 2x2x2 Rubik's Cube is designed such that the upper bound on the number of (quarter-turn) moves between any two states is 14 ("God's Number"). This serves as a natural bound for model checking the system, as a solution to the cube must reachable within 14 moves. By creating an accurate (i.e. verified) model of the Rubik's Cube, bounded model checking with a bound of 14 steps from the initial state shall produce a solution for each possible configuration of the cube.

4.3 2x2x2 Model

Initially, nuXmv [10] seemed to be the best choice for the Rubik's Cube model, as it was the symbolic and bounded model checking tool discussed in class. However, for reasons outlined below, this approach was eventually abandoned in favor of using a model created in C and checked via the C Bounded Model Checker (CBMC) [5]. This model was easier to debug and clearer to the writer, though it tended to produce quite large SAT queries on the back end.

4.3.1 nuXmv

The first model to be produced, excerpted below and fully included in Appendix D, explicitly enumerated each of the transitions for all possible turns, with goal states to be captured via LTL safety properties. An approach adopted for this model, and for all following, consisted of beginning the model at the solved state; thus, the expected output would be a series of moves to reach the (initial) unsolved state. This made verification of the model easier, as one could inductively argue for its correctness: if the initial state and transitions are correctly specified, then the model can only reach valid states.

As can be seen from the size and relative complexity of the full code contained in Appendix D, this model was very difficult to debug. No use of patterns were employed. Instead, the author used a virtual model of the cube and attempted to track faces using static images of the original, solved configuration. The primary goal of this first round of modeling was to produce something that was correct, though not necessarily efficient in terms of space; the initial SMV satisfied neither part of this goal.

```
1    ASSIGN
2
3       init (F0)  :=  yellow;
4       init (F1)  :=  yellow;
5       init (F2)  :=  yellow;
6       init (F3)  :=  yellow;
7
8       .
9       .
10      .
11
12      ——Front  transition  rules
13      next (F0)  :=
14        case
15          move = UP  :  L0;
16          move = RIGHT  :  U0;
17          move = FRONT  :  F3;
18          TRUE  :  F0;
19        esac;
20
21      next (F1)  :=
22        case
23          move = UP  :  L1;
24          move = LEFT  :  D1;
25          move = FRONT  :  F0;
26          TRUE  :  F1;
27        esac;
28
29      .
30      .
31      .
```

Excerpt_RubiksCube_2x2x2.smv

4.3.2   C

Recognizing the difficulties with capturing the moves in the first modeling attempt, a functioning simulator in C seemed a better way to capture and debug the transition relation, before returning to the XMV model. This simulator could be captured much more clearly and succinctly, with easy tools to debug via standard output (stdout), as shown in the excerpt below. Unlike in the SMV model, it is much easier to generalize the effects of any rotation; they simply consist of a clockwise rotation of the facelets on the rotating face, as well as a clockwise "swap" of the connected facelets on the four sides adjacent to the current side. By capturing this for each of the faces (and thus their associated rotations), the rotateCCW() function could be written. A clockwise rotation is

then just three counterclockwise rotations. From this "adjacencyList", it is easy to read how a counterclockwise rotation will affect the facelets: for a rotation of the FRONT face, the R0 ("Right 0") facelet will become the color of the U1 ("Up 1") facelet, which will become the color of the L2 ("Left 2") facelet.

```c
#define WHITE 0
#define BLUE 1
#define RED 2
#define GREEN 3
#define YELLOW 4
#define ORANGE 5
#define FRONT 0
#define BACK 1
#define LEFT 2
#define RIGHT 3
#define UP 4
#define DOWN 5

int cube[6][4];
     .
     .
     .
void initAdjList(){
     adjacencyList[FRONT][0][0] = &(cube[LEFT][2]);
     adjacencyList[FRONT][0][1] = &(cube[LEFT][1]);
     adjacencyList[FRONT][1][0] = &(cube[UP][1]);
     adjacencyList[FRONT][1][1] = &(cube[UP][0]);
     adjacencyList[FRONT][2][0] = &(cube[RIGHT][0]);
     adjacencyList[FRONT][2][1] = &(cube[RIGHT][3]);
     adjacencyList[FRONT][3][0] = &(cube[DOWN][1]);
     adjacencyList[FRONT][3][1] = &(cube[DOWN][0]);
     .
     .
     .
void rotateCCW(int side){
     .
     .
     .
}
void rotateCW(int side){
     rotateCCW(side);
     rotateCCW(side);
     rotateCCW(side);
}
void main(){
     int i = 0;
     char move;
     int dir;
     resetCube();
     initAdjList();
     printCube();
     .
     .
     .
}
```

Excerpt_Rubiks_2x2x2_Interactive.c

4.4 Verification

For this model, advanced methods of verification were unavailable, initially. An online simulator was employed to compare similarly formatted cubes. First, the online simulator was put through a series of rotations, producing a final cube state. Then, the C model, shown in Appendix E, was run, and the results of the rotations were compared. Initially, each individual rotation was tested and debugged. After this, the combination of all rotations was compared, to look for correct output.
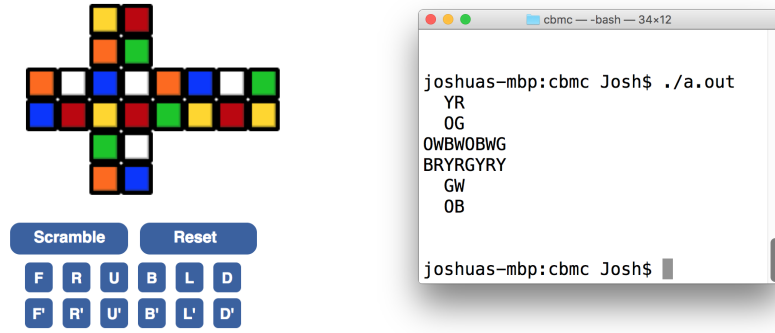

Figure 3: Comparing the move sequence 'FRUBLD'

Though simplistic, this method allowed quick elimination of simple errors (such as mis-numbering in the facelets or reversed faces in the adjacency list). After testing many sequences of rotation without apparent fault, it was determined that the transition relation (and the model itself) were correct representations of the cube.

Initially, upon completing the C model shown above, the goal was to return to the SMV model, so that nuXmv bounded model checking could be used to produce an answer. Unfortunately, this model was again difficult to capture. The primary issue was that the model seemed to grow overly complicated quite quickly, with the full state machine itself essentially being encoded. Upon this realization, the need for another model of the system was questioned. Instead, the C model itself may be used for model checking.

4.5 C Bounded Model Checker (CBMC)

The C Bounded Model Checker (CBMC) was first demonstrated by Clarke et al in 2004, as a way to check the consistency of Verilog and C models of hardware and software systems [7]. By converting ANSI-C code into an intermediate, symbolic representation, the full program could be captured in CNF form and given to a SAT solver to verify conformance to a series of assertions. Through

the use of a special set of functions to capture assumptions, assertions, and non-determinism, CBMC is able to compute whether there exists a valid execution of a C program which violates the given properties.

To prepare the model for CBMC, a few additions were made to capture the notion of the "final state" of the cube (see Appendix F). Though CBMC does provide command-line options for limiting the length of the search, as well as using built-in techniques to automatically do so (thus "Bounded"), this model instead includes a natural bound as the result of a for-loop that runs only 14 times (see "Background") and non-deterministically selects a move in each iteration. The target assertion (that the Rubik's cube is in the goal state) is then checked for each iteration.

4.5.1 CBMC Results

Upon running CBMC, with the aforementioned assertions and assumptions, the result was a SAT query with the following specifications:

Figure 4: SAT query specs for the move sequence 'FRUBLD'

This model generated a drastically larger number of variables and clauses than the direct SAT method (via SABR) outlined above. However, the CBMC approach still produced results within a reasonable time ($\sim$450s).

The results of this model checking run included a counterexample trace that could be reduced to a list of moves for solving the cube. Upon receiving this list, the moves were performed on both the online and locally simulated cubes. In both cases, the set of moves was successfully able to solve the cube.

Figure 5: Model checking results for the move sequence 'FRUBLD'

It should be noted that, should the reader choose to replicate the work given above, command line options were used with CBMC to ensure correctness. As a result of the way that the C model was written the for-loop would **always** run 16 times. Depending on the placement of the assertion, the results of the model checking run may be different:

1. If the assertion was placed within the for-loop, then the return counterexample may continue past the goal state to keep searching. This is as a result of the unrolling that occurs within CBMC. An assertion appearing within a loop is actually unrolled into a set of separate assertions, each of which is checked in the unrolled code. CBMC will continue searching until the end of the program is reached (i.e. all iterations of the loop are complete) or until all of the assertions have failed (in this case, necessarily until all iterations of the loop are complete). This is unnecessarily wasteful for the (significant) number of arrangements that are less than 16 rotations away.

2. If the assertion was placed outside of the for-loop, then, interestingly, the result is the search for a counterexample **of exactly a certain length**. In this case, for example, rather than checking for an example within 16 steps, CBMC would be searching for the target configuration after 16 moves have been nondeterministically selected and made.

To counteract this issue, CBMC includes the "–stop-on-fail" command line option. By placing the assertion within the for-loop, and using this option, the search will halt immediately upon finding a valid path to the violating state. This interesting contrast in results suggests that the model checking approach

may be find optimal solutions through an iterative approach on the for-loop (by searching for a violation within 1 move, then 2 moves, etc.). Though on the surface this appears to be true, the actual resulting counterexample is longer than the sequence of moves used to reach the target state when creating the test case ('FRUBLD').

As mentioned previously, much of the verification effort was simply by inspection. The use of CBMC, however, provided for more chances to verify the correctness of the model. Perhaps the simplest check was to just try searching from different configurations. As discussed above in "Background", it is proven that all configurations can be solved within 16 moves. As such, the lack of a counterexample for a given model checking run demonstrates that there is a flaw in the model, the asserted property, or the command line options used for CBMC. The presence of counterexamples for the tested configurations contributes towards an argument of correctness for the model. Additionally, other assertions may be made (such as checking for too many facelets of a single color, or incorrectly colored facelets along edges) to ensure that the model is correct.

4.6   3x3x3 Model

As the 2x2x2 model was successfully used to produce solutions, it was decided again that there was no need for an SMV model to perform the model checking run. Instead, again, a model in C with CBMC would be sufficient.

4.6.1   C

Unlike the case of the 2x2x2 model, the 3x3x3 model was quite simple to implement initially, taking less than an hour to create. By carefully placing the new cubes ('X5-8' for each side 'X'), the adjacency list was easily extended to capture the transition relation for the 3x3x3. Again, an interactive version of the model was created first, so that it could be compared to existing implementations for correctness. This code is included in Appendix G.

4.7   Verification

Similar to the 2x2x2 verification effort, the first round simply consisted of inspecting correctness compared to an online simulator of the cube. Each individual rotation was checked, before checking a sequence of several rotations. Errors were corrected by printing the state of the cube after each rotation, in an attempt to locate at which point the transition relation had failed.

Again, this method revealed simple errors in the model. As an example, initially, some of the center facelets would move; given the transition relation intended, these facelets should remain stationary regardless of the rotations occurring around them. The transition relation was appropriately fixed to prevent this

from happening.
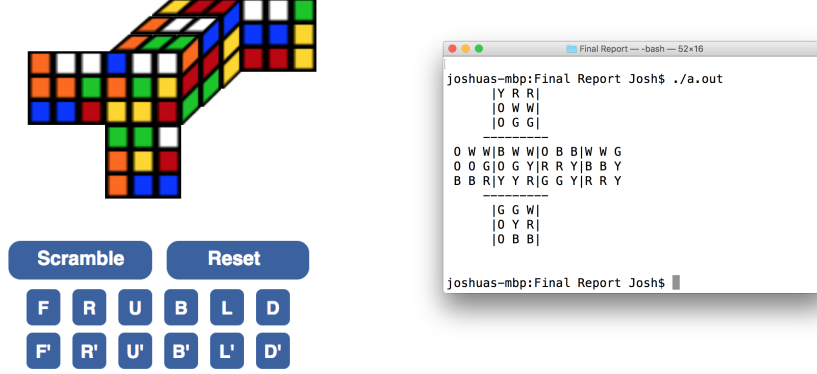
Figure 6: Comparing the move sequence 'FRUBLD'

Due to the structure of a 3x3x3 Rubik's Cube, a 2x2x2 cube may be simulated by simply disregarding the center and edge (but not corner) cubes. As a result of this property, an additional verification step was to perform the series of moves found in the first model checking run on both the online and locally simulated cubes. The result should be that the corner cubes appear solved (though the remaining cubes do not).

4.8 Initial CBMC Results

Again, the C model was appropriately annotated (see Appendix H) to allow for CBMC to locate a counterexample trace. It should be noted that, for a 3x3x3 cube, God's Number is 26 quarter turns. As a result, the for-loop is appropriately changed to account for this. CBMC was then run on this model, in the pursuit of an appropriate counterexample trace. Unfortunately, the conversion of this model to a SAT query was much less successful (see Figure 7).

Given the significant number of variables ($\sim$17 million) and clauses ($\sim$93 million) generated by CBMC, a solution was incalculable within any reasonable amount of time. In order for this method to be useful, a closer look was, and still is, needed at the inner workings of CBMC and Minisat, the underlying SAT solver called by the model checker.

4.9 "Under the hood": Minisat and CBMC

In pursuit of answers as to what optimizations might be made to the 3x3x3 model, the original papers for Minisat [9] and CBMC [7], [8] were consulted.

```
● ● ●                Final Report — cbmc --stop-on-fail Rubiks_3x3x3.c — 99×27
Unwinding loop rotateCCW.4 iteration 5 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 6 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 7 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 8 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 9 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.5 iteration 5 file Rubiks_3x3x3.c line 252 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 1 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 2 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 3 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 4 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 5 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 6 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 7 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 8 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.4 iteration 9 file Rubiks_3x3x3.c line 253 function rotateCCW thread 0
Unwinding loop rotateCCW.5 iteration 6 file Rubiks_3x3x3.c line 252 function rotateCCW thread 0
Unwinding loop main.0 iteration 26 file Rubiks_3x3x3.c line 290 function main thread 0
size of program expression: 81371 steps
simple slicing removed 58 assignments
Generated 26 VCC(s), 26 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
17077271 variables, 93027230 clauses
```
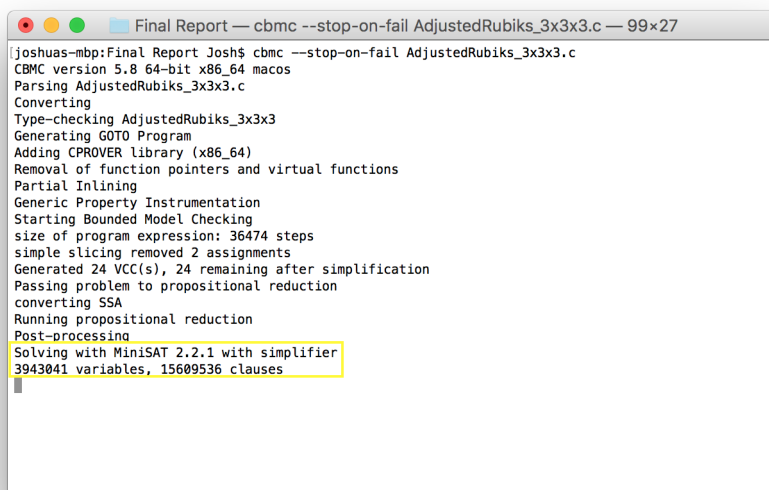
Figure 7: Initial SAT query specs for the move sequence 'FRUBLD'

Upon reviewing the CBMC paper, it was clear that the conversion heavily favored unrolled loops, an absence of pointers, and the avoidance of arrays. For pointers, the number of CNF clauses and variables grew exponentially, as a clause was generated to represent not only the actual assignment of the pointer given in the code, but also for every possible assignment based on the given datatype. Similarly, assignments to array variables generated a significant number of additional variables and clauses, to capture not only that an assignment had been made to the correct cell, but also that no assignments had been made to the other cells. Finally, loop unrolling required a series of intermediate steps, creating several GOTO statements and conditionals.

The authors of Minisat were also very transparent in the workings of their software. Minisat was designed to be intentionally clear and simple to interested developers, but was also meant to be general in the set of problems for which it is applicable. As such, several of the heuristics are not necessarily tuned to solve Rubik's Cubes. While the exact effects on this problem were not studied closely for this work, fine-tuning of multiple constants used within the source code of Minisat could improve overall performance. For example, Minisat disregards learned clauses after they go unused for a certain amount of time. It is possible that, in the case of this particular problem, these learned clauses may be valuable, and the constant could be adjusted appropriately (at the cost of memory during the software execution).

4.10 Revised 3x3x3 C Model & Final CBMC Results

To compensate for the potential sources of overhead mentioned above, the 3x3x3 model was modified to be as amenable to CNF translation as possible: all loops were unrolled, all pointers were eliminated, and all arrays were turned into individual, disconnected variables. As well, the number of function calls was reduced (as this generated additional statements in the form of GOTOs in the CNF conversion). The result was a significantly larger model (included in the repository for this project). As expected, upon running CBMC on the new model, drastically better results were observed:

Figure 8: Adjusted SAT query specs for the move sequence 'FRUBLD'

Unfortunately, though the number of variables was reduced by a factor of nearly 4 and the number of clauses by a factor of nearly 6, the reduction was not sufficient to produce a solution within a reasonable amount of time. This was somewhat shocking as the number of variables and clauses was on par with that of the 2x2x2 Rubik's Cube model checking instance. The likely cause of this discrepancy may be either the lack of fine-tuning for Minisat and CBMC, or because the complexity of the generated clauses is such that a solution cannot be computed as quickly.

One final adjustment to the 3x3x3 model was made (with the modified source available in the repository for this project), in order to see what effect it would have on the computation. Rather than unrolling every loop for the model, the

loop in main was left as it appeared in the original version of the model. CBMC was then run on this model to check for a solution:


```
● ● ●      Final Report — cbmc --stop-on-fail Adjusted2Rubiks_3x3x3.c — 99×27
Unwinding loop main.0 iteration 10 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 11 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 12 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 13 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 14 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 15 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 16 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 17 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 18 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 19 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 20 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 21 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 22 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 23 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 24 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 25 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
Unwinding loop main.0 iteration 26 file Adjusted2Rubiks_3x3x3.c line 422 function main thread 0
size of program expression: 78536 steps
simple slicing removed 5 assignments
Generated 26 VCC(s), 26 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
278348 variables, 380641 clauses
```

Figure 9: Second Adjusted SAT query specs for the move sequence 'FRUBLD'

Unexpectedly, the number of variables and clauses was reduced even further. This placed the SAT query for the 3x3x3 within two orders of magnitude of the SABR query generated above. That being said, a solution was still not computed in sufficient time. Again, it is possible that tool optimizations, the lack of model optimizations, and/or the complexity of the generated CNF clauses are to blame.

4.11 Conclusion & Future Work

Though, initially, SMV seemed like an appropriate modeling language for this problem, the actual act of modeling proved to be to complicated and difficult to bug. Alternatively, using C proved to be relatively straightforward. This approach may have simply shifted the focus from developing a correct model (as was the issue in SMV) to instead optimizing a correct model (as was the issue in C). Either way, the final C model proved to be very effective for a 2x2x2 Rubik's Cube, producing a solution in approximately seven minutes, while it was not appropriate for a 3x3x3 cube. The attempts made at optimizing code significantly reduced the size of the SAT query generated for the problem, but still the query proved to be too large for the machine used to produce a solution within reasonable time.

As discussed previously, the code for both CBMC and Minisat are publicly available, with scholarly works discussing how exactly each works. Future attempts at using a bounded model checking approach for solving Rubik's Cubes may need to focus on optimizing each of these tools appropriately for the problem. A thorough comparison of the generated SAT queries by SABR and CBMC might provide a hint as to how exactly the CBMC ANSI-C conversion could be further improved. Study of the heuristics used in Minisat, as well as the learning process used, could indicate how numeric constants and algorithms could be altered. Additionally, the work on SAT solver development included in the Minisat paper provides sufficient information such that a specialized "RubikSAT" could be developed specifically to solve this problem.

5   Final Conclusions

Though all three methods were shown to be valid solutions to the problem of solving a 2x2x2 Rubik's Cube, success was much more limited when expanded to a 3x3x3. In the case of model checking in Spin, memory issues prevented this from calculating a solution. Potential future work on this method may attempt to expand the available memory to a point that makes the model checker practically useful. Similarly, though CBMC was able to produce useful solutions for a 2x2x2 cube, the computation never terminated for calculating the results of a 3x3x3 cube. Future work in this case appears promising, focused on optimizing Minisat and CBMC's symbolic representation of ANSI-C. Finally, the direct SAT solver approach, via SABR, appeared the most successful. Though it was not able to compute solutions in all cases, it was the only method to do so for any 3x3x3 configuration. Given the number of variables and clauses generated, it is unclear what optimizations may exist. That being said, a Rubik's Cube focused SAT solver may have better luck tackling this problem.

Appendix A

Table 1. Extended miniSat 2x2 Rubik's cube results

Run Number	SAT	Parse time	Restarts	Conflicts /sec	Decisions /sec	Conflict literals	Memory used	CPU time
0	Y	0.01 s	307	24015	42318	16.40% deleted	18.83 MB	4.776 s
1	Y	0.01 s	1341	21611	32719	15.89% deleted	25.42 MB	29.368 s
2	N	0.01 s	1532	24952	37551	23.72% deleted	24.79 MB	28.036 s
3	Y	0.01 s	2044	20364	30679	18.07% deleted	26.29 MB	46.368 s
4	Y	0.01 s	24574	13853	18578	35.33% deleted	80.81 MB	1186.84 s
5	Y	0.01 s	236	26659	44732	13.62% deleted	12.32 MB	2.812 s

Table 2. Extended CaDiCaL 2x2 Rubik's cube results

Run Number	SAT	Parse time	Restarts	Conflicts /sec	Decisions /sec	Conflict literals	Memory used	CPU time
0	Y	0.01 s	9813	390697	11103.13	7.63% deleted	15.67 MB	35.67 s
1	Y	0.01 s	4693	13438.41	53381.47	7.35% deleted	9.10 MB	11.59 s
2	N	0.01 s	3167	16655.43	57761.16	7.81% deleted	8.47 MB	7.89 s
3	Y	0.01 s	29175	5824.01	21442.38	11.11% deleted	21.96 MB	160.65 s
4	Y	0.01 s	NA	NA	NA	NA	NA	20000 s
5	Y	0.01 s	2172	18536.71	74321.68	6.05% deleted	6.77 MB	4.00 s

Appendix B

```promela
/*colors: 1=blue, 2=yellow, 3= green, 4=white, 5=red, 6=pink */
byte a1=1, a2=1, a3=1, a4=1, b1=2, b2=2, b3=2, b4=2, c1=3, c2=3, c3
    =3, c4=3, d1=4, d2=4, d3=4, d4=4, e1=5, e2=5, e3=5, e4=5, f1=6,
    f2=6, f3=6, f4=6;
byte tmp1=0, tmp2=0, tmp3=0, tmp4=0, tmp5=0, tmp6=0, tmp7=0, tmp8
    =0, tmp9=0, tmp10=0, tmp11=0, tmp12=0;
#define ASIDE (a1==6 && a2==6 && a3==4 && a4==2)
#define BSIDE (b1==3 && b2==3 && b3==1 && b4==1)
#define CSIDE (c1==5 && c2==5 && c3==4 && c4==2)
#define DSIDE (d1==3 && d2==1 && d3==3 && d4==1)
#define ESIDE (e1==4 && e2==2 && e3==5 && e4==5)
#define FSIDE (f1==4 && f2==2 && f3==6 && f4==6)
#define FINISH (ASIDE && BSIDE && CSIDE && DSIDE && ESIDE)
active proctype rubiks()
{
    do

    :: assert (!FINISH) -> atomic{
        tmp1=a2;
        tmp2=a4;
        tmp3=b2;
        tmp4=b4;
        tmp5=c1;
        tmp6=c3;
        tmp7=d2;
        tmp8=d4;
        tmp9=e1;
        tmp10=e2;
        tmp11=e3;
        tmp12=e4;
        a2=tmp7;
        a4=tmp8;
        b2=tmp1;
        b4=tmp2;
        c1=tmp4;
        c3=tmp3;
        d2=tmp6;
        d4=tmp5;
        e2=tmp9;
        e4=tmp10;
        e3=tmp12;
        e1=tmp11;
        printf("Right")
        }

    :: assert (!FINISH)->atomic{
        tmp1=a1;
        tmp2=a3;
        tmp3=b1;
        tmp4=b3;
        tmp5=c2;
        tmp6=c4;
        tmp7=d1;
        tmp8=d3;
        tmp9=f1;
```

```
 53        tmp10=f2 ;
 54        tmp11=f3 ;
 55        tmp12=f4 ;
 56        a1=tmp7 ;
 57        a3=tmp8 ;
 58        b1=tmp1 ;
 59        b3=tmp2 ;
 60        c4=tmp3 ;
 61        c2=tmp4 ;
 62        d1=tmp6 ;
 63        d3=tmp5 ;
 64        f3=tmp9 ;
 65        f4=tmp11 ;
 66        f2=tmp12 ;
 67        f1=tmp10 ;
 68        printf ("Left")
 69        }
 70
 71    :: assert (!FINISH)->atomic{
 72        tmp1=a1 ;
 73        tmp2=a2 ;
 74        tmp3=e1 ;
 75        tmp4=e2 ;
 76        tmp5=c1 ;
 77        tmp6=c2 ;
 78        tmp7=f1 ;
 79        tmp8=f2 ;
 80        tmp9=b1 ;
 81        tmp10=b2 ;
 82        tmp11=b3 ;
 83        tmp12=b4 ;
 84        a1=tmp3 ;
 85        a2=tmp4 ;
 86        e1=tmp5 ;
 87        e2=tmp6 ;
 88        c1=tmp7 ;
 89        c2=tmp8 ;
 90        f1=tmp1 ;
 91        f2=tmp2 ;
 92        b2=tmp9 ;
 93        b4=tmp10 ;
 94        b3=tmp12 ;
 95        b1=tmp11 ;
 96        printf ("Top")
 97        }
 98
 99    :: assert (!FINISH)->atomic{
100        tmp1=a3 ;
101        tmp2=a4 ;
102        tmp3=e3 ;
103        tmp4=e4 ;
104        tmp5=c3 ;
105        tmp6=c4 ;
106        tmp7=f3 ;
107        tmp8=f4 ;
108        tmp9=d1 ;
109        tmp10=d2 ;
```

```
110          tmp11=d3 ;
111          tmp12=d4 ;
112          a3=tmp3 ;
113          a4=tmp4 ;
114          e3=tmp5 ;
115          e4=tmp6 ;
116          c3=tmp7 ;
117          c4=tmp8 ;
118          f3=tmp1 ;
119          f4=tmp2 ;
120          d4=tmp11 ;
121          d2=tmp12 ;
122          d1=tmp10 ;
123          d3=tmp9 ;
124          printf("Bottom")
125          }
126
127    :: assert (!FINISH)−>atomic{
128          tmp1=b3 ;
129          tmp2=b4 ;
130          tmp3=f2 ;
131          tmp4=f4 ;
132          tmp5=e1 ;
133          tmp6=e3 ;
134          tmp7=d1 ;
135          tmp8=d2 ;
136          tmp9=a1 ;
137          tmp10=a2 ;
138          tmp11=a3 ;
139          tmp12=a4 ;
140          b4=tmp3 ;
141          b3=tmp4 ;
142          d2=tmp5 ;
143          d1=tmp6 ;
144          f2=tmp7 ;
145          f4=tmp8 ;
146          a2=tmp9 ;
147          a4=tmp10 ;
148          a1=tmp11 ;
149          a3=tmp12 ;
150          printf("Front")
151          }
152
153    :: assert (!FINISH)−>atomic{
154          tmp1=b1 ;
155          tmp2=b2 ;
156          tmp3=f1 ;
157          tmp4=f3 ;
158          tmp5=e2 ;
159          tmp6=e4 ;
160          tmp7=d3 ;
161          tmp8=d4 ;
162          tmp9=c1 ;
163          tmp10=c2 ;
164          tmp11=c3 ;
165          tmp12=c4 ;
166          b2=tmp3 ;
```

```
167          b1=tmp4;
168          e2=tmp1;
169          e4=tmp2;
170          d3=tmp6;
171          d4=tmp5;
172          f1=tmp7;
173          f3=tmp8;
174          c3=tmp9;
175          c1=tmp10;
176          c4=tmp11;
177          c2=tmp12;
178          printf("Back")
179          }
180      od
181 }
182 od
183 }
```

sivets_model.pml

Sabr 3x3 Rubik's Cube model

```
1  # SABR
2  # solve 3x3 rubiks cube
3
4  Sym{ b o w r g y }
5
6  Board{
7  .....         b1 b2 b3;
8  .....         b4 .  b6;
9  .....         b7 b8 b9;
10 r1 r2 r3      y1 y2 y3     o1 o2 o3     w1 w2 w3;
11 r4 .  r6      y4 .  y6     o4 .  o6     w4 .  w6;
12 r7 r8 r9      y7 y8 y9     o7 o8 o9     w7 w8 w9;
13 .....         g1 g2 g3;
14 .....         g4 .  g6;
15 .....         g7 g8 g9;
16 }
17
18 Start{
19          y g b;
20          g   g;
21          o b w;
22 r o g    w r b    o y y    o r b;
23 b   w    o   b    y   b    o   w;
24 b w w    b y g    y r g    r o r;
25          r r o;
26          g   w;
27          w y y;
28 }
29
30 End{
31          b b b;
32          b   b;
33          b b b;
34 r r r    y y y        o o o    w w w;
```

28

```
35 r    r    y    y           o    o    w    w;
36 r  r  r    y  y  y          o  o  o    w  w  w;
37            g  g  g;
38            g       g;
39            g  g  g;
40 }
41
42 Trans  Clock : Side {
43
44      t1  t2  t3 ;
45
46 l3    f1  f2  f3       r1 ;
47 l2    f4       f6      r2 ;
48 l1    f7  f8  f9       r3 ;
49
50      b1  b2  b3 ;
51 =>
52      l1  l2  l3 ;
53
54 b1    f7  f4  f1       t1 ;
55 b2    f8       f2      t2 ;
56 b3    f9  f6  f3       t3 ;
57
58      r3  r2  r1 ;
59 }
60
61 Trans  CounterClock : Side {
62
63      t1  t2  t3 ;
64
65 l3    f1  f2  f3       r1 ;
66 l2    f4       f6      r2 ;
67 l1    f7  f8  f9       r3 ;
68
69      b1  b2  b3 ;
70 =>
71      r1  r2  r3 ;
72
73 t3    f3  f6  f9       b3 ;
74 t2    f2       f8      b2 ;
75 t1    f1  f4  f7       b1 ;
76
77      l3  l2  l1 ;
78 }
79
80 DesObj  Front : Side {
81      b7  b8  b9 ;
82
83 r3    y1  y2  y3       o1 ;
84 r6    y4       y6      o4 ;
85 r9    y7  y8  y9       o7 ;
86
87      g1  g2  g3 ;
88 }
89
90 DesObj  Top : Side {
91      w3  w2  w1 ;
```

```
r1    b1  b2  b3        o3 ;
r2    b4      b6        o2 ;
r3    b7  b8  b9        o1 ;

      y1  y2  y3 ;
}

DesObj  Back : Side {
      b3  b2  b1 ;

o3    w1  w2  w3        r1 ;
o6    w4      w6        r4 ;
o9    w7  w8  w9        r6 ;

      g8  g7  g6 ;
}

DesObj  Bottom : Side {
      y7  y8  y9 ;

r9    g1  g2  g3        o7 ;
r8    g4      g6        o8 ;
r7    g7  g8  g9        o9 ;

      w9  w8  w7 ;
}

DesObj  Left : Side {
      b1  b4  b7 ;

w3    r1  r2  r3        y1 ;
w6    r4      r6        y4 ;
w9    r7  r8  r9        y7 ;

      g7  g4  g1 ;
}

DesObj  Right : Side {
      b9  b6  b3 ;

y3    o1  o2  o3        w1 ;
y6    o4      o6        w4 ;
y9    o7  o8  o9        w7 ;

      g3  g6  g9 ;
}
```

```
1  /−−
2          2x2x2  Rubik's  Cube  Model
3
4          Addressing  particular  locations  on  cube:
5              TRF  :  Cubelet  in  Top,  Right,  Front  position
6              TRB  :  Cubelet  in  Top,  Right,  Back  position
7              TLF  :  Cubelet  in  Top,  Left,  Front  position
8              TLB  :  Cubelet  in  Top,  Left,  Back  position
9              DRF  :  Cubelet  in  Bottom,  Right,  Front  position
10             DRB  :  Cubelet  in  Bottom,  Right,  Back  position
11             DLF  :  Cubelet  in  Bottom,  Left,  Front  position
12             DLB  :  Cubelet  in  Bottom,  Left,  Back  position
13
14         Numbering  for  particular  cubes:
15             1  :  Cubelet  with  blue,  red,  white  facelets
16             2  :  Cubelet  with  blue,  red,  yellow  facelets
17             3  :  Cubelet  with  blue,  orange,  white  facelets
18             4  :  Cubelet  with  blue,  orange,  yellow  facelets
19             5  :  Cubelet  with  green,  red,  white  facelets
20             6  :  Cubelet  with  green,  red,  yellow  facelets
21             7  :  Cubelet  with  green,  orange,  white  facelets
22             8  :  Cubelet  with  green,  orange,  yellow  facelets
23
24
25
26 −−/
27
28
29
30
31 MODULE main
32   VAR
33   −−Move
34     move  :  {UP, DOWN, LEFT, RIGHT, FRONT, BACK};
35   −−Front  Side
36     F0  :  {green,  yellow,  white,  blue,  red,  orange};
37     F1  :  {green,  yellow,  white,  blue,  red,  orange};
38     F2  :  {green,  yellow,  white,  blue,  red,  orange};
39     F3  :  {green,  yellow,  white,  blue,  red,  orange};
40   −−Back  Side
41     B0  :  {green,  yellow,  white,  blue,  red,  orange};
42     B1  :  {green,  yellow,  white,  blue,  red,  orange};
43     B2  :  {green,  yellow,  white,  blue,  red,  orange};
44     B3  :  {green,  yellow,  white,  blue,  red,  orange};
45   −−Left  Side
46     L0  :  {green,  yellow,  white,  blue,  red,  orange};
47     L1  :  {green,  yellow,  white,  blue,  red,  orange};
48     L2  :  {green,  yellow,  white,  blue,  red,  orange};
49     L3  :  {green,  yellow,  white,  blue,  red,  orange};
50   −−Right  Side
51     R0  :  {green,  yellow,  white,  blue,  red,  orange};
52     R1  :  {green,  yellow,  white,  blue,  red,  orange};
53     R2  :  {green,  yellow,  white,  blue,  red,  orange};
54     R3  :  {green,  yellow,  white,  blue,  red,  orange};
55   −−Up  Side
```

```
56      U0 : {green, yellow, white, blue, red, orange};
57      U1 : {green, yellow, white, blue, red, orange};
58      U2 : {green, yellow, white, blue, red, orange};
59      U3 : {green, yellow, white, blue, red, orange};
60    --Down side
61      D0 : {green, yellow, white, blue, red, orange};
62      D1 : {green, yellow, white, blue, red, orange};
63      D2 : {green, yellow, white, blue, red, orange};
64      D3 : {green, yellow, white, blue, red, orange};
65
66    LTLSPEC G ! ((F0 = blue) & (F1 = blue) & (F2 = blue) & (F3 = blue
         ));
67
68    ASSIGN
69
70      init(F0) := yellow;
71      init(F1) := yellow;
72      init(F2) := yellow;
73      init(F3) := yellow;
74
75      init(U0) := blue;
76      init(U1) := blue;
77      init(U2) := blue;
78      init(U3) := blue;
79
80      init(R0) := red;
81      init(R1) := red;
82      init(R2) := red;
83      init(R3) := red;
84
85      init(L0) := orange;
86      init(L1) := orange;
87      init(L2) := orange;
88      init(L3) := orange;
89
90      init(B0) := white;
91      init(B1) := white;
92      init(B2) := white;
93      init(B3) := white;
94
95      init(D0) := green;
96      init(D1) := green;
97      init(D2) := green;
98      init(D3) := green;
99
100     --Front transition rules
101     next(F0) :=
102       case
103         move = UP : L0;
104         move = RIGHT : U0;
105         move = FRONT : F3;
106         TRUE : F0;
107       esac;
108
109     next(F1) :=
110       case
111         move = UP : L1;
```

```
112        move = LEFT : D1;
113        move = FRONT : F0;
114        TRUE : F1;
115      esac;
116
117    next(F2) :=
118      case
119        move = DOWN : R2;
120        move = LEFT : D2;
121        move = FRONT : F1;
122        TRUE : F2;
123      esac;
124
125    next(F3) :=
126      case
127        move = DOWN : R3;
128        move = RIGHT : U3;
129        move = FRONT : F2;
130        TRUE : F3;
131      esac;
132
133    --Right transition rules
134
135    next(R0) :=
136      case
137        move = UP : F0;
138        move = BACK : U1;
139        move = RIGHT : R3;
140        TRUE : R0;
141      esac;
142
143    next(R1) :=
144      case
145        move = UP : F1;
146        move = FRONT : B1;
147        move = RIGHT : R0;
148        TRUE : R1;
149      esac;
150
151    next(R2) :=
152      case
153        move = DOWN : B2;
154        move = FRONT : B2;
155        move = RIGHT : R1;
156        TRUE : R2;
157      esac;
158
159    next(R3) :=
160      case
161        move = DOWN : B3;
162        move = BACK : U0;
163        move = RIGHT : R2;
164        TRUE : R3;
165      esac;
166
167    --Up transition rules
168
```

```
169    next (U0) :=
170      case
171        move = RIGHT : B0;
172        move = BACK : L3;
173        move = UP : U3;
174        TRUE : U0;
175      esac;
176
177    next (U1) :=
178      case
179        move = LEFT : F1;
180        move = BACK : L0;
181        move = UP : U0;
182        TRUE : U1;
183      esac;
184
185    next (U2) :=
186      case
187        move = LEFT : F2;
188        move = FRONT : R1;
189        move = UP : U1;
190        TRUE : U2;
191      esac;
192
193    next (U3) :=
194      case
195        move = RIGHT : B3;
196        move = FRONT : R2;
197        move = UP : U2;
198        TRUE : U3;
199      esac;
200
201    --Left transition rules
202    next (L0) :=
203      case
204        move = BACK : D0;
205        move = DOWN : F2;
206        move = LEFT : L3;
207        TRUE : L0;
208      esac;
209
210    next (L1) :=
211      case
212        move = FRONT : B2;
213        move = DOWN : F3;
214        move = LEFT : L0;
215        TRUE : L1;
216      esac;
217
218    next (L2) :=
219      case
220        move = FRONT : B2;
221        move = UP : B2;
222        move = LEFT : L1;
223        TRUE : L2;
224      esac;
225
```

```
226    next (L3) :=
227       case
228         move = BACK : D3;
229         move = UP : B3;
230         move = LEFT : L2;
231         TRUE : L3;
232       esac;
233
234       --Down transition rules
235       next (D0) :=
236          case
237            move = BACK : R0;
238            move = RIGHT : F3;
239            move = DOWN : D3;
240            TRUE : D0;
241          esac;
242
243       next (D1) :=
244          case
245            move = FRONT : L1;
246            move = RIGHT : F0;
247            move = DOWN : D0;
248            TRUE : D1;
249          esac;
250
251       next (D2) :=
252          case
253            move = FRONT : L2;
254            move = LEFT : B1;
255            move = DOWN : D1;
256            TRUE : D2;
257          esac;
258
259       next (D3) :=
260          case
261            move = BACK : R3;
262            move = LEFT : B2;
263            move = DOWN : D2;
264            TRUE : D3;
265          esac;
266
267       --Back transition rules
268       next (B0) :=
269          case
270            move = DOWN : L0;
271            move = RIGHT : D1;
272            move = BACK : B3;
273            TRUE : B0;
274          esac;
275
276       next (B1) :=
277          case
278            move = DOWN : L1;
279            move = LEFT : U1;
280            move = BACK : B0;
281            TRUE : B1;
282          esac;
```

```
283
284        next(B2) :=
285          case
286            move = LEFT : U2;
287            move = UP : R0;
288            move = BACK : B1;
289            TRUE : B2;
290          esac;
291
292        next(B3) :=
293          case
294            move = RIGHT : D2;
295            move = UP : R3;
296            move = BACK : B2;
297            TRUE : B3;
298          esac;
299
300      FAIRNESS TRUE
```

RubiksCube_2x2x2.smv

Appendix E: 2x2x2 C Model

```c
#include <stdio.h>
#include <assert.h>

#define WHITE 0
#define BLUE 1
#define RED 2
#define GREEN 3
#define YELLOW 4
#define ORANGE 5

#define FRONT 0
#define BACK 1
#define LEFT 2
#define RIGHT 3
#define UP 4
#define DOWN 5

int cube[6][4];

int* adjacencyList[6][4][2];

const char side_names[6]={'W', 'B', 'R', 'G', 'Y', 'O'};

void resetCube(){
  cube[FRONT][0]=GREEN;
  cube[FRONT][1]=GREEN;
  cube[FRONT][2]=GREEN;
  cube[FRONT][3]=GREEN;
  cube[BACK][0]=BLUE;
  cube[BACK][1]=BLUE;
  cube[BACK][2]=BLUE;
  cube[BACK][3]=BLUE;
  cube[LEFT][0]=ORANGE;
  cube[LEFT][1]=ORANGE;
  cube[LEFT][2]=ORANGE;
  cube[LEFT][3]=ORANGE;
  cube[RIGHT][0]=RED;
  cube[RIGHT][1]=RED;
  cube[RIGHT][2]=RED;
  cube[RIGHT][3]=RED;
  cube[UP][0]=WHITE;
  cube[UP][1]=WHITE;
  cube[UP][2]=WHITE;
  cube[UP][3]=WHITE;
  cube[DOWN][0]=YELLOW;
  cube[DOWN][1]=YELLOW;
  cube[DOWN][2]=YELLOW;
  cube[DOWN][3]=YELLOW;
}

void initAdjList(){
  adjacencyList[FRONT][0][0] = &(cube[LEFT][3-1]);
  adjacencyList[FRONT][0][1] = &(cube[LEFT][2-1]);
  adjacencyList[FRONT][1][0] = &(cube[UP][2-1]);
  adjacencyList[FRONT][1][1] = &(cube[UP][1-1]);
```

```
56    adjacencyList[FRONT][2][0]  = &(cube[RIGHT][1−1]);
57    adjacencyList[FRONT][2][1]  = &(cube[RIGHT][4−1]);
58    adjacencyList[FRONT][3][0]  = &(cube[DOWN][2−1]);
59    adjacencyList[FRONT][3][1]  = &(cube[DOWN][1−1]);
60
61    adjacencyList[BACK][0][0]  = &(cube[DOWN][3−1]);
62    adjacencyList[BACK][0][1]  = &(cube[DOWN][4−1]);
63    adjacencyList[BACK][1][0]  = &(cube[RIGHT][2−1]);
64    adjacencyList[BACK][1][1]  = &(cube[RIGHT][3−1]);
65    adjacencyList[BACK][2][0]  = &(cube[UP][3−1]);
66    adjacencyList[BACK][2][1]  = &(cube[UP][4−1]);
67    adjacencyList[BACK][3][0]  = &(cube[LEFT][4−1]);
68    adjacencyList[BACK][3][1]  = &(cube[LEFT][1−1]);
69
70    adjacencyList[LEFT][0][0]  = &(cube[BACK][3−1]);
71    adjacencyList[LEFT][0][1]  = &(cube[BACK][2−1]);
72    adjacencyList[LEFT][1][0]  = &(cube[UP][3−1]);
73    adjacencyList[LEFT][1][1]  = &(cube[UP][2−1]);
74    adjacencyList[LEFT][2][0]  = &(cube[FRONT][1−1]);
75    adjacencyList[LEFT][2][1]  = &(cube[FRONT][4−1]);
76    adjacencyList[LEFT][3][0]  = &(cube[DOWN][1−1]);
77    adjacencyList[LEFT][3][1]  = &(cube[DOWN][4−1]);
78
79    adjacencyList[RIGHT][0][0]  = &(cube[DOWN][2−1]);
80    adjacencyList[RIGHT][0][1]  = &(cube[DOWN][3−1]);
81    adjacencyList[RIGHT][1][0]  = &(cube[FRONT][2−1]);
82    adjacencyList[RIGHT][1][1]  = &(cube[FRONT][3−1]);
83    adjacencyList[RIGHT][2][0]  = &(cube[UP][4−1]);
84    adjacencyList[RIGHT][2][1]  = &(cube[UP][1−1]);
85    adjacencyList[RIGHT][3][0]  = &(cube[BACK][4−1]);
86    adjacencyList[RIGHT][3][1]  = &(cube[BACK][1−1]);
87
88    adjacencyList[UP][0][0]  = &(cube[FRONT][1−1]);
89    adjacencyList[UP][0][1]  = &(cube[FRONT][2−1]);
90    adjacencyList[UP][1][0]  = &(cube[LEFT][1−1]);
91    adjacencyList[UP][1][1]  = &(cube[LEFT][2−1]);
92    adjacencyList[UP][2][0]  = &(cube[BACK][1−1]);
93    adjacencyList[UP][2][1]  = &(cube[BACK][2−1]);
94    adjacencyList[UP][3][0]  = &(cube[RIGHT][1−1]);
95    adjacencyList[UP][3][1]  = &(cube[RIGHT][2−1]);
96
97    adjacencyList[DOWN][0][0]  = &(cube[RIGHT][4−1]);
98    adjacencyList[DOWN][0][1]  = &(cube[RIGHT][3−1]);
99    adjacencyList[DOWN][1][0]  = &(cube[BACK][4−1]);
100   adjacencyList[DOWN][1][1]  = &(cube[BACK][3−1]);
101   adjacencyList[DOWN][2][0]  = &(cube[LEFT][4−1]);
102   adjacencyList[DOWN][2][1]  = &(cube[LEFT][3−1]);
103   adjacencyList[DOWN][3][0]  = &(cube[FRONT][4−1]);
104   adjacencyList[DOWN][3][1]  = &(cube[FRONT][3−1]);
105 }
106
107 void rotateCCW(int side){
108   int tempCube[6][4];
109   int i = 0;
110   int j = 0;
111
112   for(j = 0; j < 6; j++){
```

```
113        for ( i = 0;  i  <  4;  i++){
114            tempCube [ j ] [ i ]  =  cube [ j ] [ i ] ;
115        }
116    }
117
118
119    for ( i = 0;  i  <  4;  i++){
120        *(&(tempCube [ 0 ] [ 0 ] )  +  ( adjacencyList [ side ] [ i ] [ 0 ]  −  &(cube
                [ 0 ] [ 0 ] ) ) )  =  *( adjacencyList [ side ] [ ( i +1)%4] [ 0 ] ) ;
121        *(&(tempCube [ 0 ] [ 0 ] )  +  ( adjacencyList [ side ] [ i ] [ 1 ]  −  &(cube
                [ 0 ] [ 0 ] ) ) )  =  *( adjacencyList [ side ] [ ( i +1)%4] [ 1 ] ) ;
122        tempCube [ side ] [ i ]  =  cube [ side ] [ ( i +1) % 4 ] ;
123    }
124
125    for ( j = 0;  j  <  6;  j++){
126        for ( i = 0;  i  <  4;  i++){
127            cube [ j ] [ i ]  =  tempCube [ j ] [ i ] ;
128        }
129    }
130 }
131
132 void rotateCW( int  side ){
133    rotateCCW( side ) ;
134    rotateCCW( side ) ;
135    rotateCCW( side ) ;
136 }
137
138 void  printCube ( ){
139    printf (" %c%c      \n" ,  side_names [ cube [UP] [ 2 ] ] , side_names [ cube [UP
            ] [ 3 ] ] ) ;
140    printf (" %c%c      \n" ,  side_names [ cube [UP] [ 1 ] ] , side_names [ cube [UP
            ] [ 0 ] ] ) ;
141    printf ("%c%c%c%c%c%c%c%c\n" ,  side_names [ cube [LEFT ] [ 0 ] ] , side_names
            [ cube [LEFT ] [ 1 ] ] , side_names [ cube [FRONT] [ 0 ] ] , side_names [ cube [
            FRONT] [ 1 ] ] , side_names [ cube [RIGHT] [ 0 ] ] , side_names [ cube [RIGHT
            ] [ 1 ] ] , side_names [ cube [BACK] [ 0 ] ] , side_names [ cube [BACK] [ 1 ] ] ) ;
142    printf ("%c%c%c%c%c%c%c%c\n" ,  side_names [ cube [LEFT ] [ 3 ] ] , side_names
            [ cube [LEFT ] [ 2 ] ] , side_names [ cube [FRONT] [ 3 ] ] , side_names [ cube [
            FRONT] [ 2 ] ] , side_names [ cube [RIGHT] [ 3 ] ] , side_names [ cube [RIGHT
            ] [ 2 ] ] , side_names [ cube [BACK] [ 3 ] ] , side_names [ cube [BACK] [ 2 ] ] ) ;
143    printf (" %c%c      \n" ,  side_names [ cube [DOWN] [ 0 ] ] , side_names [ cube [
            DOWN] [ 1 ] ] ) ;
144    printf (" %c%c      \n" ,  side_names [ cube [DOWN] [ 3 ] ] , side_names [ cube [
            DOWN] [ 2 ] ] ) ;
145    printf ("\n\n") ;
146 }
147
148 void  main ( ){
149    int  i = 0;
150    char  move ;
151    int  dir ;
152
153    resetCube ( ) ;
154    initAdjList ( ) ;
155    printCube ( ) ;
156
157    for ( i = 0;  i  <  26;  i++){
```

```c
158        if(FRUBLD) break;
159
160        scanf("%1c%1d", &move, &dir);
161
162        switch(move){
163            case '0':
164              move='F';
165              break;
166            case '1':
167              move='B';
168              break;
169            case '2':
170              move='L';
171              break;
172            case '3':
173              move='R';
174              break;
175            case '4':
176              move='U';
177              break;
178            case '5':
179              move='D';
180              break;
181            default:
182              break;
183        }
184
185        switch(move){
186            case 'F':
187                (dir == 0) ? rotateCW(FRONT) : rotateCCW(FRONT);
188              break;
189            case 'B':
190                (dir == 0) ? rotateCW(BACK) : rotateCCW(BACK);
191              break;
192            case 'L':
193                (dir == 0) ? rotateCW(LEFT) : rotateCCW(LEFT);
194              break;
195            case 'R':
196                (dir == 0) ? rotateCW(RIGHT) : rotateCCW(RIGHT);
197              break;
198            case 'U':
199                (dir == 0) ? rotateCW(UP) : rotateCCW(UP);
200              break;
201            case 'D':
202                (dir == 0) ? rotateCW(DOWN) : rotateCCW(DOWN);
203              break;
204        }
205        printCube();
206    }
207 }
```

Rubiks_2x2x2_Interactive.c

Appendix F: CBMC Annotated 2x2x2 C Model

```c
#include <stdio.h>
#include <assert.h>

#define WHITE 0
#define BLUE 1
#define RED 2
#define GREEN 3
#define YELLOW 4
#define ORANGE 5

#define FRONT 0
#define BACK 1
#define LEFT 2
#define RIGHT 3
#define UP 4
#define DOWN 5

//Captures the initial (solved) state of the cube by defining what
    each of the facelets on the cube should look like
#define INITIAL_CONFIGURATION cube[FRONT][0]==GREEN&&cube[FRONT
    ][1]==GREEN&&cube[FRONT][2]==GREEN&&cube[FRONT][3]==GREEN&&cube
    [BACK][0]==BLUE&&cube[BACK][1]==BLUE&&cube[BACK][2]==BLUE&&cube
    [BACK][3]==BLUE&&cube[LEFT][0]==ORANGE&&cube[LEFT][1]==ORANGE&&
    cube[LEFT][2]==ORANGE&&cube[LEFT][3]==ORANGE&&cube[RIGHT][0]==
    RED&&cube[RIGHT][1]==RED&&cube[RIGHT][2]==RED&&cube[RIGHT][3]==
    RED&&cube[UP][0]==WHITE&&cube[UP][1]==WHITE&&cube[UP][2]==WHITE
    &&cube[UP][3]==WHITE&&cube[DOWN][0]==YELLOW&&cube[DOWN][1]==
    YELLOW&&cube[DOWN][2]==YELLOW&&cube[DOWN][3]==YELLOW

//Captures the targeted end state
#define FRUBLD cube[FRONT][0]==BLUE&&cube[FRONT][1]==WHITE&&cube[
    FRONT][2]==RED&&cube[FRONT][3]==YELLOW&&cube[BACK][0]==WHITE&&
    cube[BACK][1]==GREEN&&cube[BACK][2]==YELLOW&&cube[BACK][3]==RED
    &&cube[LEFT][0]==ORANGE&&cube[LEFT][1]==WHITE&&cube[LEFT][2]==
    RED&&cube[LEFT][3]==BLUE&&cube[RIGHT][0]==ORANGE&&cube[RIGHT
    ][1]==BLUE&&cube[RIGHT][2]==YELLOW&&cube[RIGHT][3]==GREEN&&cube
    [UP][0]==GREEN&&cube[UP][1]==ORANGE&&cube[UP][2]==YELLOW&&cube[
    UP][3]==RED&&cube[DOWN][0]==GREEN&&cube[DOWN][1]==WHITE&&cube[
    DOWN][2]==BLUE&&cube[DOWN][3]==ORANGE

int cube[6][4];

int* adjacencyList[6][4][2];

const char side_names[6]={'W', 'B', 'R', 'G', 'Y', 'O'};

int nondet_next_move();
int nondet_next_dir();

void resetCube(){
    cube[FRONT][0]=GREEN;
    cube[FRONT][1]=GREEN;
    cube[FRONT][2]=GREEN;
    cube[FRONT][3]=GREEN;
    cube[BACK][0]=BLUE;
```

```
39    cube[BACK][1]=BLUE;
40    cube[BACK][2]=BLUE;
41    cube[BACK][3]=BLUE;
42    cube[LEFT][0]=ORANGE;
43    cube[LEFT][1]=ORANGE;
44    cube[LEFT][2]=ORANGE;
45    cube[LEFT][3]=ORANGE;
46    cube[RIGHT][0]=RED;
47    cube[RIGHT][1]=RED;
48    cube[RIGHT][2]=RED;
49    cube[RIGHT][3]=RED;
50    cube[UP][0]=WHITE;
51    cube[UP][1]=WHITE;
52    cube[UP][2]=WHITE;
53    cube[UP][3]=WHITE;
54    cube[DOWN][0]=YELLOW;
55    cube[DOWN][1]=YELLOW;
56    cube[DOWN][2]=YELLOW;
57    cube[DOWN][3]=YELLOW;
58  }
59
60  void initAdjList(){
61    adjacencyList[FRONT][0][0]  = &(cube[LEFT][2]);
62    adjacencyList[FRONT][0][1]  = &(cube[LEFT][1]);
63    adjacencyList[FRONT][1][0]  = &(cube[UP][1]);
64    adjacencyList[FRONT][1][1]  = &(cube[UP][0]);
65    adjacencyList[FRONT][2][0]  = &(cube[RIGHT][0]);
66    adjacencyList[FRONT][2][1]  = &(cube[RIGHT][3]);
67    adjacencyList[FRONT][3][0]  = &(cube[DOWN][1]);
68    adjacencyList[FRONT][3][1]  = &(cube[DOWN][0]);
69
70    adjacencyList[BACK][0][0]  = &(cube[DOWN][2]);
71    adjacencyList[BACK][0][1]  = &(cube[DOWN][3]);
72    adjacencyList[BACK][1][0]  = &(cube[RIGHT][1]);
73    adjacencyList[BACK][1][1]  = &(cube[RIGHT][2]);
74    adjacencyList[BACK][2][0]  = &(cube[UP][2]);
75    adjacencyList[BACK][2][1]  = &(cube[UP][3]);
76    adjacencyList[BACK][3][0]  = &(cube[LEFT][3]);
77    adjacencyList[BACK][3][1]  = &(cube[LEFT][0]);
78
79    adjacencyList[LEFT][0][0]  = &(cube[BACK][2]);
80    adjacencyList[LEFT][0][1]  = &(cube[BACK][1]);
81    adjacencyList[LEFT][1][0]  = &(cube[UP][2]);
82    adjacencyList[LEFT][1][1]  = &(cube[UP][1]);
83    adjacencyList[LEFT][2][0]  = &(cube[FRONT][0]);
84    adjacencyList[LEFT][2][1]  = &(cube[FRONT][3]);
85    adjacencyList[LEFT][3][0]  = &(cube[DOWN][0]);
86    adjacencyList[LEFT][3][1]  = &(cube[DOWN][3]);
87
88    adjacencyList[RIGHT][0][0]  = &(cube[DOWN][1]);
89    adjacencyList[RIGHT][0][1]  = &(cube[DOWN][2]);
90    adjacencyList[RIGHT][1][0]  = &(cube[FRONT][1]);
91    adjacencyList[RIGHT][1][1]  = &(cube[FRONT][2]);
92    adjacencyList[RIGHT][2][0]  = &(cube[UP][3]);
93    adjacencyList[RIGHT][2][1]  = &(cube[UP][0]);
94    adjacencyList[RIGHT][3][0]  = &(cube[BACK][3]);
95    adjacencyList[RIGHT][3][1]  = &(cube[BACK][0]);
```

```
 96
 97    adjacencyList[UP][0][0]  = &(cube[FRONT][0]);
 98    adjacencyList[UP][0][1]  = &(cube[FRONT][1]);
 99    adjacencyList[UP][1][0]  = &(cube[LEFT][0]);
100    adjacencyList[UP][1][1]  = &(cube[LEFT][1]);
101    adjacencyList[UP][2][0]  = &(cube[BACK][0]);
102    adjacencyList[UP][2][1]  = &(cube[BACK][1]);
103    adjacencyList[UP][3][0]  = &(cube[RIGHT][0]);
104    adjacencyList[UP][3][1]  = &(cube[RIGHT][1]);
105
106    adjacencyList[DOWN][0][0]  = &(cube[RIGHT][3]);
107    adjacencyList[DOWN][0][1]  = &(cube[RIGHT][2]);
108    adjacencyList[DOWN][1][0]  = &(cube[BACK][3]);
109    adjacencyList[DOWN][1][1]  = &(cube[BACK][2]);
110    adjacencyList[DOWN][2][0]  = &(cube[LEFT][3]);
111    adjacencyList[DOWN][2][1]  = &(cube[LEFT][2]);
112    adjacencyList[DOWN][3][0]  = &(cube[FRONT][3]);
113    adjacencyList[DOWN][3][1]  = &(cube[FRONT][2]);
114
115 }
116
117
118
119 void rotateCCW(int side){
120    int tempCube[6][4];
121    int i = 0;
122    int j = 0;
123
124    for(j = 0; j < 6; j++){
125      for(i = 0; i < 4; i++){
126        tempCube[j][i] = cube[j][i];
127      }
128    }
129
130
131   for(i = 0; i < 4; i++){
132      *(&(tempCube[0][0]) + (adjacencyList[side][i][0] - &(cube
             [0][0]))) = *(adjacencyList[side][(i+1)%4][0]);
133      *(&(tempCube[0][0]) + (adjacencyList[side][i][1] - &(cube
             [0][0]))) = *(adjacencyList[side][(i+1)%4][1]);
134      tempCube[side][i] = cube[side][(i+1) % 4];
135    }
136
137    for(j = 0; j < 6; j++){
138      for(i = 0; i < 4; i++){
139        cube[j][i] = tempCube[j][i];
140      }
141    }
142 }
143
144 void rotateCW(int side){
145    rotateCCW(side);
146    rotateCCW(side);
147    rotateCCW(side);
148 }
149
150 void printCube(){
```

```c
151    printf(" %c%c     \n", side_names[cube[UP][2]], side_names[cube[UP
           ][3]]);
152    printf(" %c%c     \n", side_names[cube[UP][1]], side_names[cube[UP
           ][0]]);
153    printf("%c%c%c%c%c%c%c%c\n", side_names[cube[LEFT][0]], side_names
           [cube[LEFT][1]], side_names[cube[FRONT][0]], side_names[cube[
           FRONT][1]], side_names[cube[RIGHT][0]], side_names[cube[RIGHT
           ][1]], side_names[cube[BACK][0]], side_names[cube[BACK][1]]);
154    printf("%c%c%c%c%c%c%c%c\n", side_names[cube[LEFT][3]], side_names
           [cube[LEFT][2]], side_names[cube[FRONT][3]], side_names[cube[
           FRONT][2]], side_names[cube[RIGHT][3]], side_names[cube[RIGHT
           ][2]], side_names[cube[BACK][3]], side_names[cube[BACK][2]]);
155    printf(" %c%c     \n", side_names[cube[DOWN][0]], side_names[cube[
           DOWN][1]]);
156    printf(" %c%c     \n", side_names[cube[DOWN][3]], side_names[cube[
           DOWN][2]]);
157    printf("\n\n");
158 }
159
160 void main(){
161    int i = 0;
162    int move;
163    int dir;
164
165    resetCube();
166    initAdjList();
167
168    __CPROVER_assume(INITIAL_CONFIGURATION);
169
170    for(i = 0; i < 14; i++){
171        move = nondet_next_move();
172        dir = nondet_next_dir();
173
174        __CPROVER_assume(move >= 0 && move <= 5);
175        __CPROVER_assume(dir == 0 || dir == 1);
176
177        if(dir == 1){
178          rotateCCW(move);
179        }else if(dir == 0){
180          rotateCW(move);
181        }
182
183        __CPROVER_assert(!(FRUBLD), "REACHED GOAL");
184    }
185
186
187 }
```

Rubiks_2x2x2.c

Appendix G: 3x3x3 C Model

```c
#include <stdio.h>
#include <assert.h>

#define WHITE 0
#define BLUE 1
#define RED 2
#define GREEN 3
#define YELLOW 4
#define ORANGE 5

#define FRONT 0
#define BACK 1
#define LEFT 2
#define RIGHT 3
#define UP 4
#define DOWN 5

int cube[6][9];

int* adjacencyList[6][4][3];

const char side_names[6]={'W', 'B', 'R', 'G', 'Y', 'O'};

void resetCube(){
  cube[FRONT][0]=GREEN;
  cube[FRONT][1]=GREEN;
  cube[FRONT][2]=GREEN;
  cube[FRONT][3]=GREEN;
  cube[FRONT][4]=GREEN;
  cube[FRONT][5]=GREEN;
  cube[FRONT][6]=GREEN;
  cube[FRONT][7]=GREEN;
  cube[FRONT][8]=GREEN;
  cube[BACK][0]=BLUE;
  cube[BACK][1]=BLUE;
  cube[BACK][2]=BLUE;
  cube[BACK][3]=BLUE;
  cube[BACK][4]=BLUE;
  cube[BACK][5]=BLUE;
  cube[BACK][6]=BLUE;
  cube[BACK][7]=BLUE;
  cube[BACK][8]=BLUE;
  cube[LEFT][0]=ORANGE;
  cube[LEFT][1]=ORANGE;
  cube[LEFT][2]=ORANGE;
  cube[LEFT][3]=ORANGE;
  cube[LEFT][4]=ORANGE;
  cube[LEFT][5]=ORANGE;
  cube[LEFT][6]=ORANGE;
  cube[LEFT][7]=ORANGE;
  cube[LEFT][8]=ORANGE;
  cube[RIGHT][0]=RED;
  cube[RIGHT][1]=RED;
  cube[RIGHT][2]=RED;
  cube[RIGHT][3]=RED;
```

45

```
56    cube[RIGHT][4]=RED;
57    cube[RIGHT][5]=RED;
58    cube[RIGHT][6]=RED;
59    cube[RIGHT][7]=RED;
60    cube[RIGHT][8]=RED;
61    cube[UP][0]=WHITE;
62    cube[UP][1]=WHITE;
63    cube[UP][2]=WHITE;
64    cube[UP][3]=WHITE;
65    cube[UP][4]=WHITE;
66    cube[UP][5]=WHITE;
67    cube[UP][6]=WHITE;
68    cube[UP][7]=WHITE;
69    cube[UP][8]=WHITE;
70    cube[DOWN][0]=YELLOW;
71    cube[DOWN][1]=YELLOW;
72    cube[DOWN][2]=YELLOW;
73    cube[DOWN][3]=YELLOW;
74    cube[DOWN][4]=YELLOW;
75    cube[DOWN][5]=YELLOW;
76    cube[DOWN][6]=YELLOW;
77    cube[DOWN][7]=YELLOW;
78    cube[DOWN][8]=YELLOW;
79  }
80
81  void initAdjList(){
82    adjacencyList[FRONT][0][0]  = &(cube[LEFT][2]);
83    adjacencyList[FRONT][0][1]  = &(cube[LEFT][5]);
84    adjacencyList[FRONT][0][2]  = &(cube[LEFT][1]);
85    adjacencyList[FRONT][1][0]  = &(cube[UP][1]);
86    adjacencyList[FRONT][1][1]  = &(cube[UP][4]);
87    adjacencyList[FRONT][1][2]  = &(cube[UP][0]);
88    adjacencyList[FRONT][2][0]  = &(cube[RIGHT][0]);
89    adjacencyList[FRONT][2][1]  = &(cube[RIGHT][7]);
90    adjacencyList[FRONT][2][2]  = &(cube[RIGHT][3]);
91    adjacencyList[FRONT][3][0]  = &(cube[DOWN][1]);
92    adjacencyList[FRONT][3][1]  = &(cube[DOWN][4]);
93    adjacencyList[FRONT][3][2]  = &(cube[DOWN][0]);
94
95    adjacencyList[BACK][0][0]  = &(cube[DOWN][2]);
96    adjacencyList[BACK][0][1]  = &(cube[DOWN][6]);
97    adjacencyList[BACK][0][2]  = &(cube[DOWN][3]);
98    adjacencyList[BACK][1][0]  = &(cube[RIGHT][1]);
99    adjacencyList[BACK][1][1]  = &(cube[RIGHT][5]);
100   adjacencyList[BACK][1][2]  = &(cube[RIGHT][2]);
101   adjacencyList[BACK][2][0]  = &(cube[UP][2]);
102   adjacencyList[BACK][2][1]  = &(cube[UP][6]);
103   adjacencyList[BACK][2][2]  = &(cube[UP][3]);
104   adjacencyList[BACK][3][0]  = &(cube[LEFT][3]);
105   adjacencyList[BACK][3][1]  = &(cube[LEFT][7]);
106   adjacencyList[BACK][3][2]  = &(cube[LEFT][0]);
107
108   adjacencyList[LEFT][0][0]  = &(cube[BACK][2]);
109   adjacencyList[LEFT][0][1]  = &(cube[BACK][5]);
110   adjacencyList[LEFT][0][2]  = &(cube[BACK][1]);
111   adjacencyList[LEFT][1][0]  = &(cube[UP][2]);
112   adjacencyList[LEFT][1][1]  = &(cube[UP][5]);
```

```
113    adjacencyList[LEFT][1][2]  = &(cube[UP][1]);
114    adjacencyList[LEFT][2][0]  = &(cube[FRONT][0]);
115    adjacencyList[LEFT][2][1]  = &(cube[FRONT][7]);
116    adjacencyList[LEFT][2][2]  = &(cube[FRONT][3]);
117    adjacencyList[LEFT][3][0]  = &(cube[DOWN][0]);
118    adjacencyList[LEFT][3][1]  = &(cube[DOWN][7]);
119    adjacencyList[LEFT][3][2]  = &(cube[DOWN][3]);
120
121    adjacencyList[RIGHT][0][0]  = &(cube[DOWN][1]);
122    adjacencyList[RIGHT][0][1]  = &(cube[DOWN][5]);
123    adjacencyList[RIGHT][0][2]  = &(cube[DOWN][2]);
124    adjacencyList[RIGHT][1][0]  = &(cube[FRONT][1]);
125    adjacencyList[RIGHT][1][1]  = &(cube[FRONT][5]);
126    adjacencyList[RIGHT][1][2]  = &(cube[FRONT][2]);
127    adjacencyList[RIGHT][2][0]  = &(cube[UP][3]);
128    adjacencyList[RIGHT][2][1]  = &(cube[UP][7]);
129    adjacencyList[RIGHT][2][2]  = &(cube[UP][0]);
130    adjacencyList[RIGHT][3][0]  = &(cube[BACK][3]);
131    adjacencyList[RIGHT][3][1]  = &(cube[BACK][7]);
132    adjacencyList[RIGHT][3][2]  = &(cube[BACK][0]);
133
134    adjacencyList[UP][0][0]  = &(cube[FRONT][0]);
135    adjacencyList[UP][0][1]  = &(cube[FRONT][4]);
136    adjacencyList[UP][0][2]  = &(cube[FRONT][1]);
137    adjacencyList[UP][1][0]  = &(cube[LEFT][0]);
138    adjacencyList[UP][1][1]  = &(cube[LEFT][4]);
139    adjacencyList[UP][1][2]  = &(cube[LEFT][1]);
140    adjacencyList[UP][2][0]  = &(cube[BACK][0]);
141    adjacencyList[UP][2][1]  = &(cube[BACK][4]);
142    adjacencyList[UP][2][2]  = &(cube[BACK][1]);
143    adjacencyList[UP][3][0]  = &(cube[RIGHT][0]);
144    adjacencyList[UP][3][1]  = &(cube[RIGHT][4]);
145    adjacencyList[UP][3][2]  = &(cube[RIGHT][1]);
146
147    adjacencyList[DOWN][0][0]  = &(cube[RIGHT][3]);
148    adjacencyList[DOWN][0][1]  = &(cube[RIGHT][6]);
149    adjacencyList[DOWN][0][2]  = &(cube[RIGHT][2]);
150    adjacencyList[DOWN][1][0]  = &(cube[BACK][3]);
151    adjacencyList[DOWN][1][1]  = &(cube[BACK][6]);
152    adjacencyList[DOWN][1][2]  = &(cube[BACK][2]);
153    adjacencyList[DOWN][2][0]  = &(cube[LEFT][3]);
154    adjacencyList[DOWN][2][1]  = &(cube[LEFT][6]);
155    adjacencyList[DOWN][2][2]  = &(cube[LEFT][2]);
156    adjacencyList[DOWN][3][0]  = &(cube[FRONT][3]);
157    adjacencyList[DOWN][3][1]  = &(cube[FRONT][6]);
158    adjacencyList[DOWN][3][2]  = &(cube[FRONT][2]);
159
160 }
161
162
163
164 void rotateCCW(int side){
165    int tempCube[6][9];
166    int i = 0;
167    int j = 0;
168
169    for(j = 0; j < 6; j++){
```

```c
170        for(i = 0; i < 9; i++){
171          tempCube[j][i] = cube[j][i];
172        }
173      }
174
175    for(i = 0; i < 4; i++){
176        *(&(tempCube[0][0]) + (adjacencyList[side][i][0] - &(cube
                [0][0]))) = *(adjacencyList[side][(i+1)%4][0]);
177        *(&(tempCube[0][0]) + (adjacencyList[side][i][1] - &(cube
                [0][0]))) = *(adjacencyList[side][(i+1)%4][1]);
178        *(&(tempCube[0][0]) + (adjacencyList[side][i][2] - &(cube
                [0][0]))) = *(adjacencyList[side][(i+1)%4][2]);
179    }
180
181    for(i = 0; i < 3; i++){
182        tempCube[side][i] = cube[side][(i+1) % 4];
183        tempCube[side][i+4] = cube[side][i+5];
184    }
185
186    tempCube[side][3] = cube[side][0];
187    tempCube[side][7] = cube[side][4];
188
189    for(j = 0; j < 6; j++){
190        for(i = 0; i < 9; i++){
191          cube[j][i] = tempCube[j][i];
192        }
193    }
194  }
195
196  void rotateCW(int side){
197    rotateCCW(side);
198    rotateCCW(side);
199    rotateCCW(side);
200  }
201
202  void printCube(){
203    printf("        |%1c%2c%2c|      \n", side_names[cube[UP][2]],
            side_names[cube[UP][6]], side_names[cube[UP][3]]);
204    printf("        |%1c%2c%2c|      \n", side_names[cube[UP][5]],
            side_names[cube[UP][8]], side_names[cube[UP][7]]);
205    printf("        |%1c%2c%2c|      \n", side_names[cube[UP][1]],
            side_names[cube[UP][4]], side_names[cube[UP][0]]);
206    printf("         —————————      \n");
207    printf("%2c%2c%2c|%1c%2c%2c|%1c%2c%2c|%1c%2c%2c\n", side_names[
            cube[LEFT][0]], side_names[cube[LEFT][4]], side_names[cube[LEFT
            ][1]], side_names[cube[FRONT][0]], side_names[cube[FRONT][4]],
            side_names[cube[FRONT][1]], side_names[cube[RIGHT][0]],
            side_names[cube[RIGHT][4]], side_names[cube[RIGHT][1]],
            side_names[cube[BACK][0]], side_names[cube[BACK][4]],
            side_names[cube[BACK][1]]);
208    printf("%2c%2c%2c|%1c%2c%2c|%1c%2c%2c|%1c%2c%2c\n", side_names[
            cube[LEFT][7]], side_names[cube[LEFT][8]], side_names[cube[LEFT
            ][5]], side_names[cube[FRONT][7]], side_names[cube[FRONT][8]],
            side_names[cube[FRONT][5]], side_names[cube[RIGHT][7]],
            side_names[cube[RIGHT][8]], side_names[cube[RIGHT][5]],
            side_names[cube[BACK][7]], side_names[cube[BACK][8]],
            side_names[cube[BACK][5]]);
```

```c
209      printf("%2c%2c%2c|%1c%2c%2c|%1c%2c%2c|%1c%2c%2c\n", side_names[
             cube[LEFT][3]], side_names[cube[LEFT][6]], side_names[cube[LEFT
             ][2]], side_names[cube[FRONT][3]], side_names[cube[FRONT][6]],
             side_names[cube[FRONT][2]], side_names[cube[RIGHT][3]],
             side_names[cube[RIGHT][6]], side_names[cube[RIGHT][2]],
             side_names[cube[BACK][3]], side_names[cube[BACK][6]],
             side_names[cube[BACK][2]]);
210      printf("            ————————          \n");
211      printf("        |%1c%2c%2c|        \n", side_names[cube[DOWN][0]],
             side_names[cube[DOWN][4]], side_names[cube[DOWN][1]]);
212      printf("        |%1c%2c%2c|        \n", side_names[cube[DOWN][7]],
             side_names[cube[DOWN][8]], side_names[cube[DOWN][5]]);
213      printf("        |%1c%2c%2c|        \n", side_names[cube[DOWN][3]],
             side_names[cube[DOWN][6]], side_names[cube[DOWN][2]]);
214      printf("\n\n");
215  }
216
217  void main(){
218      int i = 0;
219      int move;
220      int dir;
221
222      resetCube();
223      initAdjList();
224      rotateCW(FRONT);
225      rotateCW(RIGHT);
226      rotateCW(UP);
227      rotateCW(BACK);
228      rotateCW(LEFT);
229      rotateCW(DOWN);
230      printCube();
231  }
```

Rubiks_3x3x3_Interactive.c

Appendix H: CBMC Annotated 3x3x3 C Model

```c
#include <stdio.h>
#include <assert.h>

#define WHITE 0
#define BLUE 1
#define RED 2
#define GREEN 3
#define YELLOW 4
#define ORANGE 5

#define FRONT 0
#define BACK 1
#define LEFT 2
#define RIGHT 3
#define UP 4
#define DOWN 5

#define INITIAL_CONFIGURATION cube[FRONT][0]==GREEN&&cube[FRONT
    ][1]==GREEN&&cube[FRONT][2]==GREEN&&cube[FRONT][3]==GREEN&&cube
    [FRONT][4]==GREEN&&cube[FRONT][5]==GREEN&&cube[FRONT][6]==GREEN
    &&cube[FRONT][7]==GREEN&&cube[FRONT][8]==GREEN&&cube[BACK][0]==
    BLUE&&cube[BACK][1]==BLUE&&cube[BACK][2]==BLUE&&cube[BACK][3]==
    BLUE&&cube[BACK][4]==BLUE&&cube[BACK][5]==BLUE&&cube[BACK][6]==
    BLUE&&cube[BACK][7]==BLUE&&cube[BACK][8]==BLUE&&cube[LEFT][0]==
    ORANGE&&cube[LEFT][1]==ORANGE&&cube[LEFT][2]==ORANGE&&cube[LEFT
    ][3]==ORANGE&&cube[LEFT][4]==ORANGE&&cube[LEFT][5]==ORANGE&&
    cube[LEFT][6]==ORANGE&&cube[LEFT][7]==ORANGE&&cube[LEFT][8]==
    ORANGE&&cube[RIGHT][0]==RED&&cube[RIGHT][1]==RED&&cube[RIGHT
    ][2]==RED&&cube[RIGHT][3]==RED&&cube[RIGHT][4]==RED&&cube[RIGHT
    ][5]==RED&&cube[RIGHT][6]==RED&&cube[RIGHT][7]==RED&&cube[RIGHT
    ][8]==RED&&cube[UP][0]==WHITE&&cube[UP][1]==WHITE&&cube[UP
    ][2]==WHITE&&cube[UP][3]==WHITE&&cube[UP][4]==WHITE&&cube[UP
    ][5]==WHITE&&cube[UP][6]==WHITE&&cube[UP][7]==WHITE&&cube[UP
    ][8]==WHITE&&cube[DOWN][0]==YELLOW&&cube[DOWN][1]==YELLOW&&cube
    [DOWN][2]==YELLOW&&cube[DOWN][3]==YELLOW&&cube[DOWN][4]==YELLOW
    &&cube[DOWN][5]==YELLOW&&cube[DOWN][6]==YELLOW&&cube[DOWN][7]==
    YELLOW&&cube[DOWN][8]==YELLOW
#define FRUBLD cube[FRONT][0]==BLUE&&cube[FRONT][1]==WHITE&&cube[
    FRONT][2]==RED&&cube[FRONT][3]==YELLOW&&cube[FRONT][4]==WHITE&&
    cube[FRONT][5]==YELLOW&&cube[FRONT][6]==YELLOW&&cube[FRONT
    ][7]==ORANGE&&cube[FRONT][8]==GREEN&&cube[BACK][0]==WHITE&&cube
    [BACK][1]==GREEN&&cube[BACK][2]==YELLOW&&cube[BACK][3]==RED&&
    cube[BACK][4]==WHITE&&cube[BACK][5]==YELLOW&&cube[BACK][6]==RED
    &&cube[BACK][7]==BLUE&&cube[BACK][8]==BLUE&&cube[LEFT][0]==
    ORANGE&&cube[LEFT][1]==WHITE&&cube[LEFT][2]==RED&&cube[LEFT
    ][3]==BLUE&&cube[LEFT][4]==WHITE&&cube[LEFT][5]==GREEN&&cube[
    LEFT][6]==BLUE&&cube[LEFT][7]==ORANGE&&cube[LEFT][8]==ORANGE&&
    cube[RIGHT][0]==ORANGE&&cube[RIGHT][1]==BLUE&&cube[RIGHT][2]==
    YELLOW&&cube[RIGHT][3]==GREEN&&cube[RIGHT][4]==BLUE&&cube[RIGHT
    ][5]==YELLOW&&cube[RIGHT][6]==GREEN&&cube[RIGHT][7]==RED&&cube[
    RIGHT][8]==RED&&cube[UP][0]==GREEN&&cube[UP][1]==ORANGE&&cube[
    UP][2]==WHITE&&cube[UP][3]==RED&&cube[UP][4]==GREEN&&cube[UP
    ][5]==ORANGE&&cube[UP][6]==RED&&cube[UP][7]==WHITE&&cube[UP
    ][8]==WHITE&&cube[DOWN][0]==GREEN&&cube[DOWN][1]==WHITE&&cube[
    DOWN][2]==BLUE&&cube[DOWN][3]==ORANGE&&cube[DOWN][4]==GREEN&&
```

```
           cube[DOWN][5]==RED&&cube[DOWN][6]==BLUE&&cube[DOWN][7]==ORANGE
           &&cube[DOWN][8]==YELLOW
20
21   int  cube[6][9];
22
23   int* adjacencyList[6][4][3];
24
25   const char side_names[6]={ 'W', 'B', 'R', 'G', 'Y', 'O'};
26
27   int  nondet_next_move();
28   int  nondet_next_dir();
29
30   void resetCube(){
31      cube[FRONT][0]=GREEN;
32      cube[FRONT][1]=GREEN;
33      cube[FRONT][2]=GREEN;
34      cube[FRONT][3]=GREEN;
35      cube[FRONT][4]=GREEN;
36      cube[FRONT][5]=GREEN;
37      cube[FRONT][6]=GREEN;
38      cube[FRONT][7]=GREEN;
39      cube[FRONT][8]=GREEN;
40      cube[BACK][0]=BLUE;
41      cube[BACK][1]=BLUE;
42      cube[BACK][2]=BLUE;
43      cube[BACK][3]=BLUE;
44      cube[BACK][4]=BLUE;
45      cube[BACK][5]=BLUE;
46      cube[BACK][6]=BLUE;
47      cube[BACK][7]=BLUE;
48      cube[BACK][8]=BLUE;
49      cube[LEFT][0]=ORANGE;
50      cube[LEFT][1]=ORANGE;
51      cube[LEFT][2]=ORANGE;
52      cube[LEFT][3]=ORANGE;
53      cube[LEFT][4]=ORANGE;
54      cube[LEFT][5]=ORANGE;
55      cube[LEFT][6]=ORANGE;
56      cube[LEFT][7]=ORANGE;
57      cube[LEFT][8]=ORANGE;
58      cube[RIGHT][0]=RED;
59      cube[RIGHT][1]=RED;
60      cube[RIGHT][2]=RED;
61      cube[RIGHT][3]=RED;
62      cube[RIGHT][4]=RED;
63      cube[RIGHT][5]=RED;
64      cube[RIGHT][6]=RED;
65      cube[RIGHT][7]=RED;
66      cube[RIGHT][8]=RED;
67      cube[UP][0]=WHITE;
68      cube[UP][1]=WHITE;
69      cube[UP][2]=WHITE;
70      cube[UP][3]=WHITE;
71      cube[UP][4]=WHITE;
72      cube[UP][5]=WHITE;
73      cube[UP][6]=WHITE;
74      cube[UP][7]=WHITE;
```

```
75    cube[UP][8]=WHITE;
76    cube[DOWN][0]=YELLOW;
77    cube[DOWN][1]=YELLOW;
78    cube[DOWN][2]=YELLOW;
79    cube[DOWN][3]=YELLOW;
80    cube[DOWN][4]=YELLOW;
81    cube[DOWN][5]=YELLOW;
82    cube[DOWN][6]=YELLOW;
83    cube[DOWN][7]=YELLOW;
84    cube[DOWN][8]=YELLOW;
85  }
86
87  void initAdjList(){
88    adjacencyList[FRONT][0][0]  = &(cube[LEFT][2]);
89    adjacencyList[FRONT][0][1]  = &(cube[LEFT][5]);
90    adjacencyList[FRONT][0][2]  = &(cube[LEFT][1]);
91    adjacencyList[FRONT][1][0]  = &(cube[UP][1]);
92    adjacencyList[FRONT][1][1]  = &(cube[UP][4]);
93    adjacencyList[FRONT][1][2]  = &(cube[UP][0]);
94    adjacencyList[FRONT][2][0]  = &(cube[RIGHT][0]);
95    adjacencyList[FRONT][2][1]  = &(cube[RIGHT][7]);
96    adjacencyList[FRONT][2][2]  = &(cube[RIGHT][3]);
97    adjacencyList[FRONT][3][0]  = &(cube[DOWN][1]);
98    adjacencyList[FRONT][3][1]  = &(cube[DOWN][4]);
99    adjacencyList[FRONT][3][2]  = &(cube[DOWN][0]);
100
101   adjacencyList[BACK][0][0]  = &(cube[DOWN][2]);
102   adjacencyList[BACK][0][1]  = &(cube[DOWN][6]);
103   adjacencyList[BACK][0][2]  = &(cube[DOWN][3]);
104   adjacencyList[BACK][1][0]  = &(cube[RIGHT][1]);
105   adjacencyList[BACK][1][1]  = &(cube[RIGHT][5]);
106   adjacencyList[BACK][1][2]  = &(cube[RIGHT][2]);
107   adjacencyList[BACK][2][0]  = &(cube[UP][2]);
108   adjacencyList[BACK][2][1]  = &(cube[UP][6]);
109   adjacencyList[BACK][2][2]  = &(cube[UP][3]);
110   adjacencyList[BACK][3][0]  = &(cube[LEFT][3]);
111   adjacencyList[BACK][3][1]  = &(cube[LEFT][7]);
112   adjacencyList[BACK][3][2]  = &(cube[LEFT][0]);
113
114   adjacencyList[LEFT][0][0]  = &(cube[BACK][2]);
115   adjacencyList[LEFT][0][1]  = &(cube[BACK][5]);
116   adjacencyList[LEFT][0][2]  = &(cube[BACK][1]);
117   adjacencyList[LEFT][1][0]  = &(cube[UP][2]);
118   adjacencyList[LEFT][1][1]  = &(cube[UP][5]);
119   adjacencyList[LEFT][1][2]  = &(cube[UP][1]);
120   adjacencyList[LEFT][2][0]  = &(cube[FRONT][0]);
121   adjacencyList[LEFT][2][1]  = &(cube[FRONT][7]);
122   adjacencyList[LEFT][2][2]  = &(cube[FRONT][3]);
123   adjacencyList[LEFT][3][0]  = &(cube[DOWN][0]);
124   adjacencyList[LEFT][3][1]  = &(cube[DOWN][7]);
125   adjacencyList[LEFT][3][2]  = &(cube[DOWN][3]);
126
127   adjacencyList[RIGHT][0][0]  = &(cube[DOWN][1]);
128   adjacencyList[RIGHT][0][1]  = &(cube[DOWN][5]);
129   adjacencyList[RIGHT][0][2]  = &(cube[DOWN][2]);
130   adjacencyList[RIGHT][1][0]  = &(cube[FRONT][1]);
131   adjacencyList[RIGHT][1][1]  = &(cube[FRONT][5]);
```

```c
132    adjacencyList[RIGHT][1][2] = &(cube[FRONT][2]);
133    adjacencyList[RIGHT][2][0] = &(cube[UP][3]);
134    adjacencyList[RIGHT][2][1] = &(cube[UP][7]);
135    adjacencyList[RIGHT][2][2] = &(cube[UP][0]);
136    adjacencyList[RIGHT][3][0] = &(cube[BACK][3]);
137    adjacencyList[RIGHT][3][1] = &(cube[BACK][7]);
138    adjacencyList[RIGHT][3][2] = &(cube[BACK][0]);
139
140    adjacencyList[UP][0][0] = &(cube[FRONT][0]);
141    adjacencyList[UP][0][1] = &(cube[FRONT][4]);
142    adjacencyList[UP][0][2] = &(cube[FRONT][1]);
143    adjacencyList[UP][1][0] = &(cube[LEFT][0]);
144    adjacencyList[UP][1][1] = &(cube[LEFT][4]);
145    adjacencyList[UP][1][2] = &(cube[LEFT][1]);
146    adjacencyList[UP][2][0] = &(cube[BACK][0]);
147    adjacencyList[UP][2][1] = &(cube[BACK][4]);
148    adjacencyList[UP][2][2] = &(cube[BACK][1]);
149    adjacencyList[UP][3][0] = &(cube[RIGHT][0]);
150    adjacencyList[UP][3][1] = &(cube[RIGHT][4]);
151    adjacencyList[UP][3][2] = &(cube[RIGHT][1]);
152
153    adjacencyList[DOWN][0][0] = &(cube[RIGHT][3]);
154    adjacencyList[DOWN][0][1] = &(cube[RIGHT][6]);
155    adjacencyList[DOWN][0][2] = &(cube[RIGHT][2]);
156    adjacencyList[DOWN][1][0] = &(cube[BACK][3]);
157    adjacencyList[DOWN][1][1] = &(cube[BACK][6]);
158    adjacencyList[DOWN][1][2] = &(cube[BACK][2]);
159    adjacencyList[DOWN][2][0] = &(cube[LEFT][3]);
160    adjacencyList[DOWN][2][1] = &(cube[LEFT][6]);
161    adjacencyList[DOWN][2][2] = &(cube[LEFT][2]);
162    adjacencyList[DOWN][3][0] = &(cube[FRONT][3]);
163    adjacencyList[DOWN][3][1] = &(cube[FRONT][6]);
164    adjacencyList[DOWN][3][2] = &(cube[FRONT][2]);
165
166 }
167
168
169
170 void rotateCCW(int side){
171    int tempCube[6][9];
172    int i = 0;
173    int j = 0;
174
175    for(j = 0; j < 6; j++){
176        for(i = 0; i < 9; i++){
177            tempCube[j][i] = cube[j][i];
178        }
179    }
180
181   for(i = 0; i < 4; i++){
182        *(&(tempCube[0][0]) + (adjacencyList[side][i][0] - &(cube
                [0][0]))) = *(adjacencyList[side][(i+1)%4][0]);
183        *(&(tempCube[0][0]) + (adjacencyList[side][i][1] - &(cube
                [0][0]))) = *(adjacencyList[side][(i+1)%4][1]);
184        *(&(tempCube[0][0]) + (adjacencyList[side][i][2] - &(cube
                [0][0]))) = *(adjacencyList[side][(i+1)%4][2]);
185    }
```

53

```
186
187    for(i = 0; i < 3; i++){
188        tempCube[side][i] = cube[side][(i+1) % 4];
189        tempCube[side][i+4] = cube[side][i+5];
190    }
191
192    tempCube[side][3] = cube[side][0];
193    tempCube[side][7] = cube[side][4];
194
195    for(j = 0; j < 6; j++){
196      for(i = 0; i < 9; i++){
197        cube[j][i] = tempCube[j][i];
198      }
199    }
200 }
201
202 void rotateCW(int side){
203    rotateCCW(side);
204    rotateCCW(side);
205    rotateCCW(side);
206 }
207
208 void printCube(){
209    printf("      |%1c%2c%2c|     \n", side_names[cube[UP][2]],
            side_names[cube[UP][6]], side_names[cube[UP][3]]);
210    printf("      |%1c%2c%2c|     \n", side_names[cube[UP][5]],
            side_names[cube[UP][8]], side_names[cube[UP][7]]);
211    printf("      |%1c%2c%2c|     \n", side_names[cube[UP][1]],
            side_names[cube[UP][4]], side_names[cube[UP][0]]);
212    printf("       ————————        \n");
213    printf("%2c%2c%2c|%1c%2c%2c|%1c%2c%2c|%1c%2c%2c\n", side_names[
            cube[LEFT][0]], side_names[cube[LEFT][4]], side_names[cube[LEFT
            ][1]], side_names[cube[FRONT][0]], side_names[cube[FRONT][4]],
            side_names[cube[FRONT][1]], side_names[cube[RIGHT][0]],
            side_names[cube[RIGHT][4]], side_names[cube[RIGHT][1]],
            side_names[cube[BACK][0]], side_names[cube[BACK][4]],
            side_names[cube[BACK][1]]);
214    printf("%2c%2c%2c|%1c%2c%2c|%1c%2c%2c|%1c%2c%2c\n", side_names[
            cube[LEFT][7]], side_names[cube[LEFT][8]], side_names[cube[LEFT
            ][5]], side_names[cube[FRONT][7]], side_names[cube[FRONT][8]],
            side_names[cube[FRONT][5]], side_names[cube[RIGHT][7]],
            side_names[cube[RIGHT][8]], side_names[cube[RIGHT][5]],
            side_names[cube[BACK][7]], side_names[cube[BACK][8]],
            side_names[cube[BACK][5]]);
215    printf("%2c%2c%2c|%1c%2c%2c|%1c%2c%2c|%1c%2c%2c\n", side_names[
            cube[LEFT][3]], side_names[cube[LEFT][6]], side_names[cube[LEFT
            ][2]], side_names[cube[FRONT][3]], side_names[cube[FRONT][6]],
            side_names[cube[FRONT][2]], side_names[cube[RIGHT][3]],
            side_names[cube[RIGHT][6]], side_names[cube[RIGHT][2]],
            side_names[cube[BACK][3]], side_names[cube[BACK][6]],
            side_names[cube[BACK][2]]);
216    printf("       ————————        \n");
217    printf("      |%1c%2c%2c|     \n", side_names[cube[DOWN][0]],
            side_names[cube[DOWN][4]], side_names[cube[DOWN][1]]);
218    printf("      |%1c%2c%2c|     \n", side_names[cube[DOWN][7]],
            side_names[cube[DOWN][8]], side_names[cube[DOWN][5]]);
```

```
219    printf(" |%1c%2c%2c| \n", side_names[cube[DOWN][3]],
           side_names[cube[DOWN][6]], side_names[cube[DOWN][2]]);
220    printf("\n\n");
221  }
222
223  void main(){
224    int i = 0;
225    int move;
226    int dir;
227
228    resetCube();
229    initAdjList();
230
231    __CPROVER_assume(INITIAL_CONFIGURATION);
232
233    for(i = 0; i < 26; i++){
234        move = nondet_next_move();
235        dir = nondet_next_dir();
236
237        __CPROVER_assume(move >= 0 && move <= 5);
238        __CPROVER_assume(dir == 0 || dir == 1);
239
240        if(dir == 1){
241          rotateCCW(move);
242        }else if(dir == 0){
243          rotateCW(move);
244        }
245
246        __CPROVER_assert(!(FRUBLD), "REACHED GOAL");
247    }
248  }
```

Rubiks_3x3x3.c

References

[1] 2DSim *Online 2D Pocket Cube Simulator*. https://ruwix.com/online-puzzle-simulators/2x2x2-pocket-cube-simulator.php.

[2] 3DSim *Online 3D Rubik's Cube Simulator*. https://ruwix.com/online-puzzle-simulators/3x3x3-rubiks-cube-simulator.php.

[3] Biere, A. et al. "Bounded Model Checking." *Advances in Computers*, vol. 58, 2003.

[4] *CaDiCaL. http://fmv.jku.at/cadical/*

[5] *The C Bounded Model Checker (CBMC)*. http://www.cprover.org/cbmc/.

[6] Chen, Jingchao. *Solving Rubik's Cube Using SAT Solver. Computing Research Repository - CORR, 2011.*

[7] Clarke, E. , Kroening, D. and Yorav, K. 2003. "Behavioral consistency of C and verilog programs using bounded model checking." In *Proceedings of the 40th annual Design Automation Conference (DAC '03)*. ACM, New York, NY.

[8] Clarke, E., Kroening, D., Sharygina, N. et al. Formal Methods in System Design (2004) 25: 105.

[9] Een N., Sorensson N. (2004) An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*. SAT 2003. Lecture Notes in Computer Science, vol 2919.

[10] *nuXmv Model Checker*. https://nuxmv.fbk.eu/.

[11] *Sabr. http://sabrlang.org/*

[12] SatRubiks. *Github repository. https://github.com/IowaStateAerospaceCourses-Rozier/final-project-satrubiks*