Model checking games and a genome sequence search

S M Staroletov
Polzunov Altai State Technical University, Lenin Avenue 46, Barnaul, Russia

E-mail: serg_soft@mail.ru

Abstract. The paper is considered a concept of model checking games to solve algorithmic puzzles. We describe current approaches in this field and move to a game between a user and a software model checker with the goal to provide a solution to a problem, encoded in a transition system and an LTL formula with a requirement. We show how to encode and solve some problems using this approach. Then we move to the problem of searching a pattern in a genome sequence. We implement the Z-function search method in Promela, construct the model, provide the input of real viral data, and then play a model checking game with SPIN verifier. We created a fuzzy substring search method using the non-deterministic choice operator. Based on experiments we made, we discuss that the problem to find a pattern with some deviations is only solvable using the swarm verification and hash compact methods.

1. Introduction
To foster interest in logical methods, in particular, formal verification, it is advisable to train such methods using games and puzzles. However, the same methods can be used to solve real problems. In this paper, we show the use of non-deterministic programming for the task of finding a pattern in a genome sequence.

The contributions of the paper are (a) focus on the model checking games concept (b) show how to encode real algorithms in Promela (c) discuss an effective implementation of string comparison in Promela (d) move to a fuzzy string comparison (e) apply it in the genome sequence search (f) discuss ways how to solve hard computation tasks in model checking using the swarm model checking and state hashing approaches.

2. Related work
The concept of model checking games originated from logic and theoretical model checking. The evaluation of logical formulae can be described by such games, played by two players on an arena which is formed as the product of a structure $K$ and a formula $\varphi$. One player (Verifier) attempts to prove that $\varphi$ is satisfied in $K$ while the other (Falsifier) tries to refute this [1]. Earlier, this formalism was used in [2] to play property checking games in a process calculus and modal mu logic with pre-defined rules for players’ moves. The formalism is used to study a particular logic and construct winning strategies. Recently presented [3] differential hybrid games are contests of two players, called Angel and Demon, over hybrid program $\alpha$ and property $\phi$ that is $[\alpha]\phi$ and $<\alpha>\neg\phi$ refer to complementary winning conditions ($\phi$ for Demon, $\neg\phi$ for Angel); the achievements in this theory can be used to construct cooperative hybrid systems. In this paper, we proceed to a different way: the model checking game will...
have two players (a user and a software model checker), the user declares that the system does not satisfy some formula $\phi$ and the model checker tries to refute it and provide a counter-example.

Moving to genome mathematics, an introduction to the field is done in [4]. Some methods to reduce the computational complexity of molecular biology tasks are presented in [5]. The community-approved approach to rapid sequence comparison, basic local alignment search tool (BLAST), is presented in [6].

In this paper, we apply the $Z$-function approach to pattern search and encode it in Promela language, that has not been done before.

3. Background

3.1. Model checking with SPIN

SPIN [7] is a utility for model checking the correctness of distributed software. The abbreviation SPIN stands for “Simple Promela Interpreter”. The SPIN system verifies not the programs themselves, but their models [8]. To build a model for an original parallel program or an algorithm, the verifying engineer (usually manually) builds a representation of this program in a C-like input language, called Promela (Protocol MEta-LAnguage).

The main purpose of the language is to model discrete systems and protocols [9]; nevertheless, due to its initial design, the language is capable to describe real sophisticated models of different software types. One complex model is discussed in [10] where an ODE was modeled and checked using explicit floating-point arithmetic.

In this paper, we rely on the following language features:

- the presence of arrays;
- the presence of do-while loops;
- the presence of if clause including non-deterministic choice.

As well as we use the following SPIN model checker features:

- checking of LTL properties expressed in predicates with key program variables;
- ability to generate a counter-example as a trail of visited states if the LTL property does not hold;
- optimized depth-first search (DFS);
- bitstate hashing to dramatically reduce used memory;
- ability to parallelize the model checking process using the swarm technique.

3.2. Bitstate hashing and hashcompact

In order to reduce memory for storing the states, in addition to strict (exhaustive) verification, SPIN offers hashing methods to do the checking that can visit most of the states until a hash collision occurred.

In such case, for every state of $S$ bits, a hash value of $m$ bits is computed, which is associated with a $m$ unique bit position within a large bit array of size $2^m$ [11]. For every new generated hash value, the tool inspects the current value of the bit that corresponds to the hash value, and if it is zero, set it to one. If the bit is already set, it counts this as a hash collision. Supplementary, the SPIN tool by default uses two hash functions, and stores two bits in the bit array for every state. A hash collision now requires a collision on both bits.

An alternative strategy recommended in [12] is called hashcompact. In the hashcompact method, the state descriptor is compressed from $S$ bits to 64 bits, using a single hash function. The resulting 64-bit values are then stored in a normal lookup table with collision resolution.
3.3. Swarm model checking

Swarm model checking is an approach to generate and run a bunch of verification tests (VTs) by combining three basic ideas to modify the search process [13]:

- search randomization (use different seed values for non-deterministic choices);
- search diversification (performing searches forwards or in reverse, varying hashing options);
- search parallelization (run multiple VTs in parallel).

Swarm is implemented using a pre-processor tool that generates a script to compile different VTs from an input Promela model and runs them.

3.4. SARS-CoV-2 genome: related information

In this subsection, we describe the coronavirus genome related information that corresponds to our goals. SARS-CoV-2, the coronavirus that cases COVID-19 pandemic, is having a strong influence to the world economy, led to thousands of deaths and changed plans of billions of people; in the other side, it catalyzes the processes of digitalization and puts an enormous interest to research in the sphere of biology and computational biology.

The coronavirus genome has been already decoded and is available in [14], in [15] Wu et al. analyzed the genome and found that it is 89% similar to the bat coronavirus bat-SL-CoVZC45 [16]. The viral genome is represented as a single-stranded RNA, which consists of adenine (A), guanine (G), cytosine (C) and uracil (U), but uracil symbol is often represented as thymine (T) in the sequence to do proper software support, so we have a string of 29903 nucleotides with alphabet \{A, G, C, T\}.

In figure 1, we depicted a graphical representation of the viral genome that was built using a patched version of mfold [17].

One of the main tasks in this field is the comparison of the genomes, that can help to investigate from whose animals the virus has come and which parts of them are changed due to mutations. The latter brings us to the problem of string comparison, and the comparison should be fuzzy (it should allow some deviations).

3.5. Substring search methods

In this subsection, we recall some methods of finding a pattern in a given string.

In a naive algorithm, the search for all admissible shifts is performed using a cycle in which the condition for the equality of the current characters of the string and the pattern is checked. Such an algorithm has $O(N^2)$ complexity, where $N$ corresponds to the length of the string (we will use $|string|$ for the length).

There are plenty of algorithms to do the searching more effectively (including hashing, trie, suffix automaton, Knuth–Morris–Pratt automaton). In this paper, we consider the Z-function algorithm, which
is popular in the competitive programming contests and requires $|\text{string}| + |\text{pattern}|$ additional memory, and has the complexity of $O(N)$ to build the $Z$-function and $O(N)$ to search the pattern. It is fast and requires no pointers or complex data structures so it could be implemented in such a modeling language as Promela.

The $Z$-function from string $S$ is an array $Z$, each element $Z[i]$ of which is the length of the longest common prefix between $S$ and the suffix of $S$ starting at $i$:

$$Z[i] = \max\{k : S[i..i+k] = S[0..k]\}$$  \hspace{1cm} (1)

The pseudocode to build the $Z$-function [18] in a loop through a given string and a pattern according to (1):

```c
int[] zFunction(s : string):
  int[] zf = int[|s|]
  int left = 0, right = 0
  for i = 1 to |s| - 1
    zf[i] = max(0, min(right - i, zf[i - left]))
    while i + zf[i] < |s| and
      if i + zf[i] > right
        left = i
        right = i + zf[i]
      zf[i]++
  return zf
```

The pseudocode to find a substring $\text{pattern}$ in a string $\text{text}$ using the $Z$-function [18]:

```c
int patternSearch(text: string, pattern: string):
  int[] zf = zFunction(pattern + '#' + text)
  for i = |pattern| + 1 to |text| + 1
    if zf[i] == |pattern| return i
```

Using the Promela language features, we prepared the following implementation of the $Z$-function construction:

```promela
short zf[STR_SIZE+PATTERN_SIZE+1];

inline S(j, ret) {
  if ::(j < PATTERN_SIZE) ->
    ret = pattern[j];
  ::(j == PATTERN_SIZE) ->
    ret = EPS; //#
  ::(j > PATTERN_SIZE) -> ret = text[j - PATTERN_SIZE - 1];
  fi
}

inline MAX(a, b, ret) {
  if ::(a >= b) -> ret = a;
  ::else -> ret = b;
  fi
}

inline MIN(a, b, ret) {
  if ::(a < b) -> ret = a;
  ::else -> ret = b;
}

//building the Z-function
int left = 0;
int right = 0;
int n = STR_SIZE + PATTERN_SIZE + 1;
int i = 1;
do ::(pos1 < MAX1) -> {
  printf("%d\n", i);
  int min = 0;
  int max = 0;
  int min = 0;
  int max = 0;
  zf[i] = max;
  bool isOk = true;
do ::isOk -> {
```
short s1 = 0;
S(zf[i], s1);
short s2 = 0;
S((i + zf[i]), s2);
isOk = (i + zf[i] < n) && (s1 == s2);
if ::isOk
   zf[i] = zf[i] + 1;
   ::else -> break;
else
   skip;
fi
}

It is pretty similar to the algorithmic pseudocode that we have shown before, but this implementation opens some doors to use the formal verification and model checking games for fuzzy comparing of genomes.

4. Model checking games
In this section, we describe a concept of Model checking games. In Karpov’s book [19], the method of puzzle solving by the model checker was introduced by the example of wolf, goat and cabbage problem. The idea of the puzzle is as follows:

It is necessary to transfer the both three alive to the different side of a river using series of trips in a boat that only carry two objects and a ferryman. While the heroes stay steady in the presence of the man, but there exist some restrictions while they stay alone on one and the other side of the river: the wolf can eat the goat and the goat can eat the cabbage.

A domain-specific approach to construct and solve the task is given in [20]. The task can be solved using a recursive DFS algorithm, by trying a path of transfers for different objects with these restrictions. Using model checking, it is proposed first to encode the state of the system and the rules of changing the state, then create an LTL rule that "Always the finite state will not be reached" and if the solution really exists, the model checker can find a path to the finite state and present it as a counterexample. Moreover, the state trail to the end state becomes the solution of the problem.

We describe the approach using another different simple Numeric puzzle that requires not so much coding. Let there be a number n, then:

- if it is even, divide it by 2, i.e. n \Rightarrow n / 2;
- if it is odd, multiply by 3 and add 1, i.e. n \Rightarrow 3n + 1;
- repeat the actions until n achieves 1.

If we start with the number 7, is it possible to get 1?

To solve the problem, we encode the task rules in Promela:

int N;
active proctype main() {
   N = 7;
   do ::(N % 2 == 0) -> {
      printf("n = %d, div \n", N);
      N = N / 2;
   }::else -> {
      printf("n = %d, 3n+1\n", N);
      N = 3 * N + 1;
   }
   od
   ltl check_me { [] (N != 1)}
}

We have also added the LTL clause check_me, in which we try to ensure that n will never be 1. As the solution exists, the model checker while the verification will find a path to get 1 from 7 using the
rules, and the steps how to get 1 will be printed by our printf operators in the simulation mode using a
generated counter-example trail.

Then we can move to a more sophisticated example from our tutorial in [21]. We solve the well-
known Hanoi tower puzzle:

Let there be three rods on which one can mount round disks of different sizes, and it is a rule that a
smaller disk can be put on a larger disk, but not vice versa. We assume that there are five discs. Initially,
they are all on the first rod. It is necessary to move them (all five) to the third rod, using the second as
an intermediate.

Our structure of the solution in Promela could be as the follows:

```c
#define N 5
//each rod state
byte rod1[N];
byte rod2[N];
byte rod3[N];
//disk count by each rod
byte count1, count2, count3;
byte moves;
ltl count_check
{} (count3 != 5)
}

active proctype Step()
{
count1 = N;
moves = 0;
rod1[0] = 5;
rod1[1] = 4;
rod1[2] = 3;
rod1[3] = 2;
rod1[4] = 1;
byte disk;
do
::moves == 250 -> break;
::count1 > 0 -> {
disk = rod1[count1-1];
//try 1 -> 2 rod
if
::(count2 == 0 || (count2 < N &&
    rod2[count2-1] > disk)) -> {
    printf("Disk %d from 1 to 2 \n",
    disk);
    rod2[count2] = disk;
moves++;
count1--;
count2++;
}
//try 1 -> 3; try 2 -> 1
//try 2 -> 3;
//try 3 -> 1; try 3 -> 2
od
}
```

In this snapshot, we encode the problem as a loop in which we try to move a top disk to a different
rod if it is possible with respect to the rules. Note we also encode some non-deterministic choices, to
move or not to move the disk, and it makes the possibility to check all the variants. The LTL rule "always
that we will not collect all the five disks at the last rod" is a denial of the condition for solving the
problem. In our experiment, the model checker finds the solution as a counter-example but it is not
optimal, therefore, the number of moves should also be added to the LTL property and some series of
solutions should be done.

After all, now we can describe the process of the model checking game (figure 2). Based on the task
description, the user creates a transition system just to describe steps of the task, as well as an LTL
formula to encode the inability of reaching the final state (the user is a sceptic now). Then the user starts
to play a game with the model checker. The verifier program is able to say "you are not right, the problem
is solvable, here is the counter-example" if the solution really exists and the checker is capable to solve
the task in a given time and memory constraints using its optimized DFS algorithms. Then the user can
improve the rules, change the program to add more output or properties to solve the problem more
optimally based on state trail that he got from the model checker. Then the game will continue until the
user is satisfied.
Formally, model checking games can be represented as

\[
\begin{align*}
\text{MODELCHECKING} & \vdash \lnot \phi, T \\
& \Rightarrow ((C \subseteq T) \vdash \phi) \lor \emptyset
\end{align*}
\]  

(2)

Where \( \phi \) is an LTL formula, \( T \) is a transition system, \( C \) is a counter-example as a solution of the task, part of the transition system, \( \emptyset \) here means that the verifier was unable to find a counter-example.

![Figure 2. A scheme of the model checking games.](image)

5. On the implementation of fuzzy comparison of the genome during a model checking game

5.1. Fuzzy string search
In order to do fuzzy substring search, we added the following into the last code in the section 3:

- a non-deterministic choice when we compare symbols while building the \( Z \)-function;
- a condition to limit the possible percentage of changes.

So, the comparison process (instead of testing \( s1==s1 \)) becomes:

```c
bool isEquals = false;
if 
  ::(s1 == s2) -> isEquals = true;
  ::(s1 != s2) -> isEquals = false;
  ::(s1 != s2) -> casesTotal++;
if 
  casesOk * 100 / casesTotal <= prob)
    { isEquals = true; casesOk++; }
  ::else -> isEquals = false;
fi
fi
isOk = isEquals;
```

Here \( prob \) is a given percentage probability with which we want to compare the strings.
This algorithm means that when constructing the Z-function, the equality of characters is tested and a deviation is allowed with a given probability.

5.2. Input data generation
As input strings are represented as arrays, we add the definition for the input alphabet:

```c
#define A 0
#define T 1
#define G 2
#define C 3
```

Promela does not support I/O operations, so we implemented a .fasta file (with the input genome sequence) processor and a Promela code generator, in figure 3 we show a fragment of the generated input data sequence.

Figure 3. A sample of generated input sequence.

Using such an approach, we get a bunch of lines of code, 29903 states and transitions just to fill the input data of the viral genome. Another variant could be using embedded C-code constructions to load the data according to [22], but we did not test it since we rely on pure model checking techniques.

5.3. Never claim for the model checking game
According to rules of model checking games, we should specify a negation for the rule that shows the fact of solving the puzzle. In a fuzzy genome sequence search, we specify a simple rule:

\[ G(\text{impossible} \equiv 1) \tag{3} \]

("always it is impossible to find a substring"), where variable `impossible` is the variable that is changed in a linear substring search using Z-function we built previously. Here is the code to do this search:

```c
bit impossible = 1;
int pos = 0;
do
::(i < n) -> {
if ::(zf[i] == PATTERN_SIZE) -> {
pos = i;
break;
}
::else -> skip;
fi
} ::else -> break;
```

(8)
if
::(pos >= 0) -> {
impossible = 0;
printf("We found a substring!");
}
::
else -> skip;
fi

ltl check_never
{[] (impossible == 1)

5.4. Overall structure of the solution
Summarizing previous subsections, we present the model structure to do pattern fuzzy searching using Promela and model checking (figure 4).

Figure 4. The structure of our genome search application.

The model program loads sequences, builds the Z-function and makes the substring search with the given probability of deviation. If found, the control variable is set. The model checking game here – to ask the model checker that it will never happen and its duty is to provide a counter-example with string substitutions after which the genome pattern can be found in the given genome sequence.

6. Results and their discussion
To start, we tried to do a search for a small pattern (less than 100 chars) with some mutations in a whole genome sequence string using a simple laptop with 8GB of RAM.

- After some runs in the SPIN simulation mode (it uses different random seeds) we were able to see that the substring is found.
- Exhausted verification is not feasible due to huge memory and time consumption (as we have the non-deterministic choice and large state space).
- Bitstate mode ("-DBITSTATE") with default parameters runs for some time and goes out of memory.
- Hashcompact mode ("-DHC") with default parameters has a small memory consumption but finishes without producing any counter-example.
- Swarm model checking with default swarm script runs out of memory (because it uses the bitstate mode).
- Swarm model checking using a patched swarm script to substitute the bistate mode to the hashcompact mode was able to produce a counter-example and solve the task.
To think further, a real model checking game to compare two full genomes with a given deviation rate (in this case, SARS-CoV-2 and bat-SL-CoVZC45 to prove 89% similarity) requires a lot of VTs with different randomized transitions seeds. The task here should be divided into loading the data to common memory (the same phase to all VTs) and then different Z-function calculations using the same data. It would require a custom model checker. We also see that the CPU swarm technique is not a good idea to execute a bunch of VTs, and possible GPU swarm [23] or FPGA swarm [24] should be used.

References

[1] Fischer D, Gradel E and Kaiser L 2010 Theory of Computing Systems 47 696-719
[2] Stevens P and Stirling C 1998 International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Springer) 85-101
[3] Platzer A 2017 ACM Transactions on Computational Logic TOCL 18 1-44
[4] Percus J K 2002 Mathematics of genome analysis Cambridge University Press 17
[5] Ganbaatar G, Altangerel K and Ishdorj T O 2018 International Conference on Bio-Inspired Computing: Theories and Applications (Springer) 107-19
[6] Altschul S F, Gish W, Miller W, Myers E W and Lipman D J 1990 Journal of molecular biology 215 403-10
[7] Holzmann G J 1997 IEEE Transactions on software engineering 23 279-95
[8] Staroletov S and Dubko A 2019 System Informatics 15 13-44
[9] Staroletov S, Shilov N, Zybubin V, Liakh T, Rozov A, Konyukhov I, Shilov I, Baar T and Schulte H 2019 Proc. of the Conference on Modeling and Analysis of Complex Systems and Processes (MACSPro 2019) (CEUR-WS)
[10] Staroletov S and Shilov N 2019 International Symposium on Model Checking Software (Springer) 193-207
[11] Holzmann G J 1998 Formal methods in system design 13 289-307
[12] Wolper P and Leroy D 1993 International Conference on Computer Aided Verification (Springer) 59-70
[13] Holzmann G J, Joshi R and Groce A 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (IEEE) 1-6
[14] Severe Acute Respiratory Syndrome Coronavirus 2 Isolate Wuhan-Hu-1 complete genome https://www.ncbi.nlm.nih.gov/nuccore/NC_045512.2
[15] Wu F, Zhao S, Yu B, Chen Y M, Wang W, Song Z G, Hu Y, Tao Z W, Tian J H, Pei Y Y et al. 2020 Nature 579 265-9
[16] Bat SARS-Like Coronavirus Isolate Bat-SL-Covzc45 complete genome https://www.ncbi.nlm.nih.gov/nuccore/ MG772933
[17] Zuker M, Mathews D H and Turner D H 1999 RNA Biochemistry and Biotechnology (Springer) 11-43
[18] IFMO wiki notes Z-function https://neerc.ifmo.ru/wiki/index.php?title=Z-%D1%84%D1%83%D0%BD% D0%BA%D1%86%D0%B8%D1%8F
[19] Karpov Y G 2010 Model Checking Verification of Parallel and Distributed Program Systems (Russian) ISBN 978-5-9775-0404-1
[20] Baar T 2015 Bulletin of Novosibirsk Computing Center Computer Science subseries 29-36
[21] Staroletov S 2018 Basics of Software Testing and Verification (Russian, Saint Petersburg: Lanbook) ISBN 978-5-8114-3041-3 https://lanbook.com/catalog/informatika/osnovy-testirovaniya-i-verifikatsii- programmno-g-obespecheniya/
[22] Holzmann G J 2000 International SPIN Workshop on Model Checking of Software (Springer) 131-47
[23] DeFrancisco R, Cho S, Ferdman M and Smolka S A 2019 International Symposium on Model Checking Software (Springer) 94-113
[24] Cho S, Ferdman M and Milder P 2018 FPL 435-42