Modeling the tower of Hanoi in temporal logic and finding its solution by driven deterministic reasoning

Andrea Giotti

Third version, June 2022

Introduction

The tower of Hanoi is a traditional and ancient recursive game, used also to benchmark logic programs. In most cases, the state of the system is not modeled but only the resolution strategy. When present, this state is often updated by explicit commands and not deduced from the constraints of the game itself. In this study, the game is described as a satisfiability problem of a set of specified constraints and that brings to a combinatorial set of possible resolution paths. To avoid non-determinism a classic drive has been added to the specification of the system, which gives all the necessary hints to find the solution of the game in a minimum number of moves. The overall specification has been statically and dynamically validated by deterministic automatic reasoning, so as to produce the succession of system states from the beginning of the game to its end and each step has been displayed for checking.

Constraint specification

- Each disk occupies an unique position
- If a position is occupied, it is occupied by an unique disk
- The largest disk can occupy only the base of the three rods
- Any smaller disk on higher levels of the three rods lean on a larger disk underlying
- If a move occurs, an unique disk moves for each instant of time
- Each disk which does not move preserves its current position in next instant of time
- If a disk moves from its current position of the source rod, it appears in some position of the destination rod in next instant of time
- Any disk but the smallest cannot move if it supports a smaller disk
- The game starts with all the disks stacked on the first rod

Output specification[1]

- An empty position is represented by an underscore
- A position occupied by a disk is represented by the alphabetic character corresponding to this disk

Strategy specification

- Any hint of order zero suggests to move a single disk from a source rod to a destination rod
- Any hint of higher order suggests to move all the disks from a source rod to a destination rod while using an intermediate rod as temporary storage (not explicitly modeled)
- Any hint of higher order is decomposed into an hint of lower order from a source rod to an intermediate rod, anticipated of the time required to execute this task, an hint of order zero from the source rod to a destination rod and another hint of lower order from the intermediate rod to the destination rod, delayed of the time required to execute this task
- The game starts with the first rod as source, the second as intermediate and the third as destination

1 Actual output is graphic and shown later but the format used to share the system state is textual.

Additional concepts

- A free disks is a disk which can move
- A disk belongs to any rod if it is stacked in some position of this rod
- A disk can fit a rod if there are no smaller disks belonging to this rod
- A move from a rod to another one occurs if there is an hint to do so and there is a free disk belonging to the first rod which can fit the destination rod

Design choices

- Three-valued deterministic reasoning to satisfy the specification
- Separate management of time to represent the evolution of the system state
- Separate representation of rules valid for each instant of time and facts which describe the initial conditions for execution
- Integration or temporal logic for system modeling and conventional programming for output

Adopted tools[2]

- Basic Temporal Logic selected as language for source code of the specification
- TING compiler to translate the specification to an executable temporal inference network
- TINX (single core) and TINX_MT (multiple cores) inference engines to execute the temporal inference network in real time
- GTINXSH graphic shell as testing and debugging environment
- C coding for a graphic display client interfaced by POSIX IPC to the inference engine

Notes

- The number of disks is a parameter of the specification, it has to be greater than one and it has a default value of eight
- Final conditions of the game, all the disks stacked on the third rod, have not been specified since an explicit resolution strategy is provided by the hints of the drive
- With a small addition in driving, it is easy to put the system in an infinite loop of inferences which periodically brings the system state back to the initial conditions
- Anticausal reasoning is necessary to execute this specification

Solution

Once translated previous description in BTL clauses, the compile, test and debug phases of common software development cycle can be followed to validate the specification. Constraints can be stressed by random moves and the results displayed by the custom client interface in a suitable form, while state variables are traced as textual histories by the development environment. After the addition of the drive, the game is solved by the inference engine for eight disks in 256 steps and less than 250 ms on a 2.3 GHz computer, with an overall performance of more than 20 MLIPS, but graphic output requires some time more.

2 TINX Suite is freely available on SourceForge at the link: "https://sourceforge.net/projects/temporal-inference-engine/"

hanoi.btl source code

```
/*
       The tower of Hanoi described as a satisfiability problem of a set of constraints,
       with the traditional recursive resolution algorithm to act as a drive

       Written by Andrea Giotti, revision 2, June 2022

       The variable "state(i, j)" represents j-th level of i-th rod and contains the
       character corresponding to each disk ("A"-"Z") or underscore ("_") if empty
*/

define tot = 8;

/* Constraints */

iter(disk on tot)
       {
       aux move(disk), free(disk);

       iter(base on 3)
              {
              aux stay(disk, base), fits(disk, base);

              iter(level on tot)
                     aux hanoi(disk, base, level);
              }
       }

iter(base on 3)
       iter(dest on 3)
              aux transfer(base, dest), hint(base, dest);

iter(disk on tot)
       {
       unique(hanoi(disk, pos / tot, pos % tot), pos on 3 * tot);

       ~ move(disk) --> forall(forall(hanoi(disk, base, level) == hanoi(disk, base, level) @ 1,
                                                                       level on tot), base on 3);
       ~ free(disk) --> ~ move(disk);

       iter(base on 3)
              {
              stay(disk, base) == exists(hanoi(disk, base, level), level on tot);

              when(disk is tot - 1)
                     fits(tot - 1, base);
              else
                     fits(disk, base) == ~ exists(exists(hanoi(k, base, level),
                                                  k in disk + 1 : tot - 1), level in 0 : disk);

              iter(level on tot)
                     {
                     when(disk is 0)
                            {
                            when(level in 1 : tot - 1)
                                   ~ hanoi(0, base, level);
                            }
                     else
                            when(level in 1 : tot - 1)
                                   hanoi(disk, base, level) -->
                                                 exists(hanoi(k, base, level - 1), k on disk);

                     when(disk in 0 : tot - 2)
                            when(level in 0 : tot - 2)
                                   hanoi(disk, base, level) -->
                                          (free(disk) == ~ exists(hanoi(k, base, level + 1),
                                                                       k in disk + 1 : tot - 1));
                     }

              iter(dest on 3)
                     move(disk) & transfer(base, dest) --> stay(disk, base) & stay(disk, dest) @ 1;
              }

       init hanoi(disk, 0, disk) @ 0;
       }

free(tot - 1);
```

```
iter(base on 3)
       {
       iter(level on tot)
              exists(hanoi(disk, base, level), disk on tot) -->
                                                 unique(hanoi(disk, base, level), disk on tot);

       iter(dest on 3)
              transfer(base, dest) == hint(base, dest) &
                            exists(stay(disk, base) & free(disk) & fits(disk, dest), disk on tot);
       }

exists(exists(transfer(base, dest), dest on 3), base on 3) --> unique(move(disk), disk on tot);

/* Output */

iter(disk on 27)
       iter(n on 8)
              aux symbol(disk, n);

code(symbol, "_ABCDEFGHIJKLMNOPQRSTUVWXYZ");

iter(base on 3)
       iter(level on tot)
              {
              iter(n on 8)
                     output [packed] state(base, level, 0, n);

              forall(~ hanoi(disk, base, level), disk on tot) -->
                                   forall(symbol(0, n) == state(base, level, 0, n), n on 8);

              iter(disk on tot)
                     hanoi(disk, base, level) -->
                            forall(symbol(disk + 1, n) == state(base, level, 0, n), n on 8);
              }

/* Drive */

iter(source on 3)
       iter(inter on 3)
              iter(dest on 3)
                     iter(order on tot)
                            aux drive(order, source, inter, dest);

iter(source on 3)
       iter(inter on 3)
              iter(dest on 3)
                     {
                     drive(0, source, inter, dest) --> hint(source, dest);

                     iter(order on tot - 1)
                            drive(order + 1, source, inter, dest) ==
                                   drive(order, source, dest, inter) @ - 2 ^ order &
                                   hint(source, dest) &
                                   drive(order, inter, source, dest) @ 2 ^ order;
                     }

init drive(tot - 1, 0, 1, 2) @ 2 ^ (tot - 1) - 1;
```

Additional code to loop forever

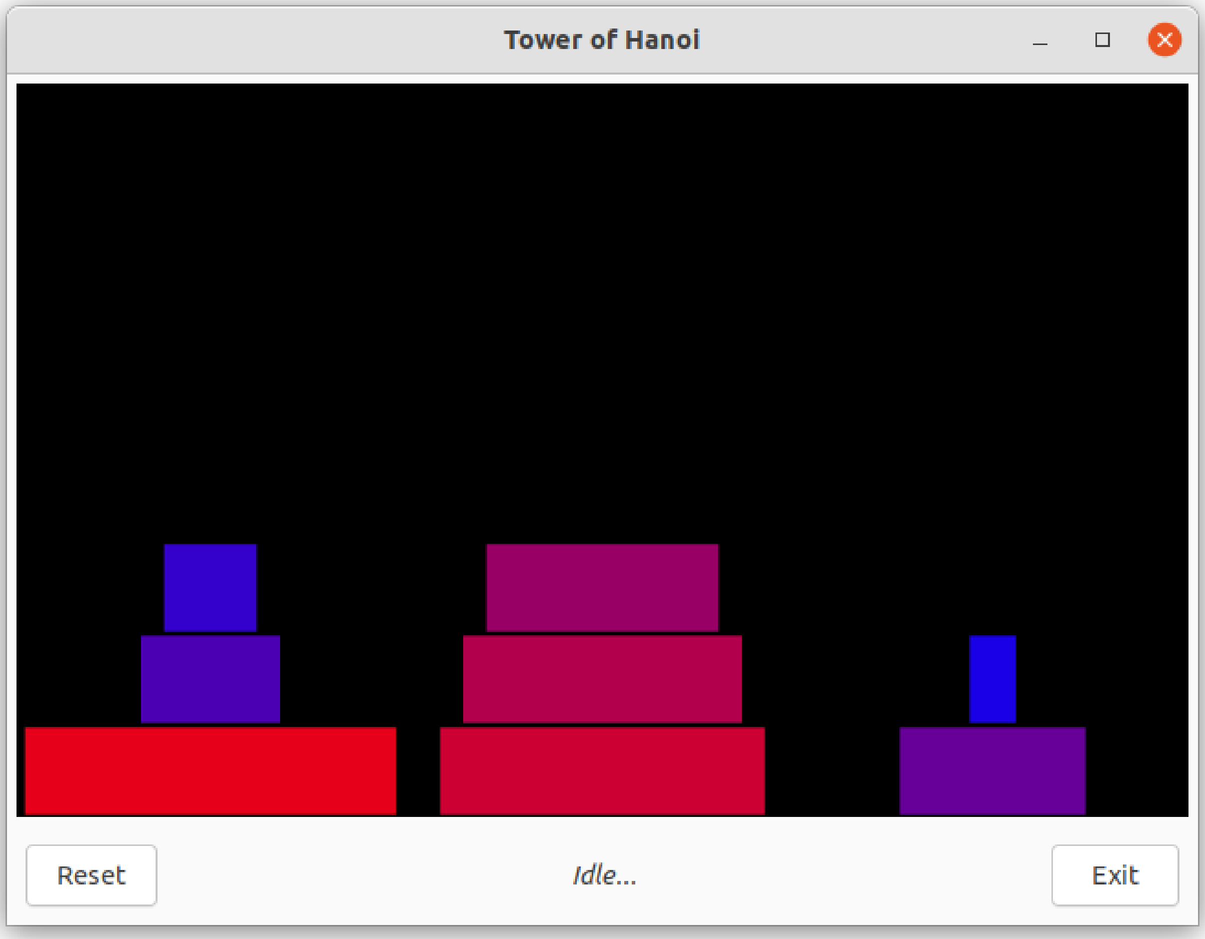
```
drive(tot - 1, 0, 1, 2) --> drive(tot - 1, 2, 0, 1) @ 2 ^ tot - 1;
drive(tot - 1, 2, 0, 1) --> drive(tot - 1, 1, 2, 0) @ 2 ^ tot - 1;
drive(tot - 1, 1, 2, 0) --> drive(tot - 1, 0, 1, 2) @ 2 ^ tot - 1;
```

Replacement code for interactive hints

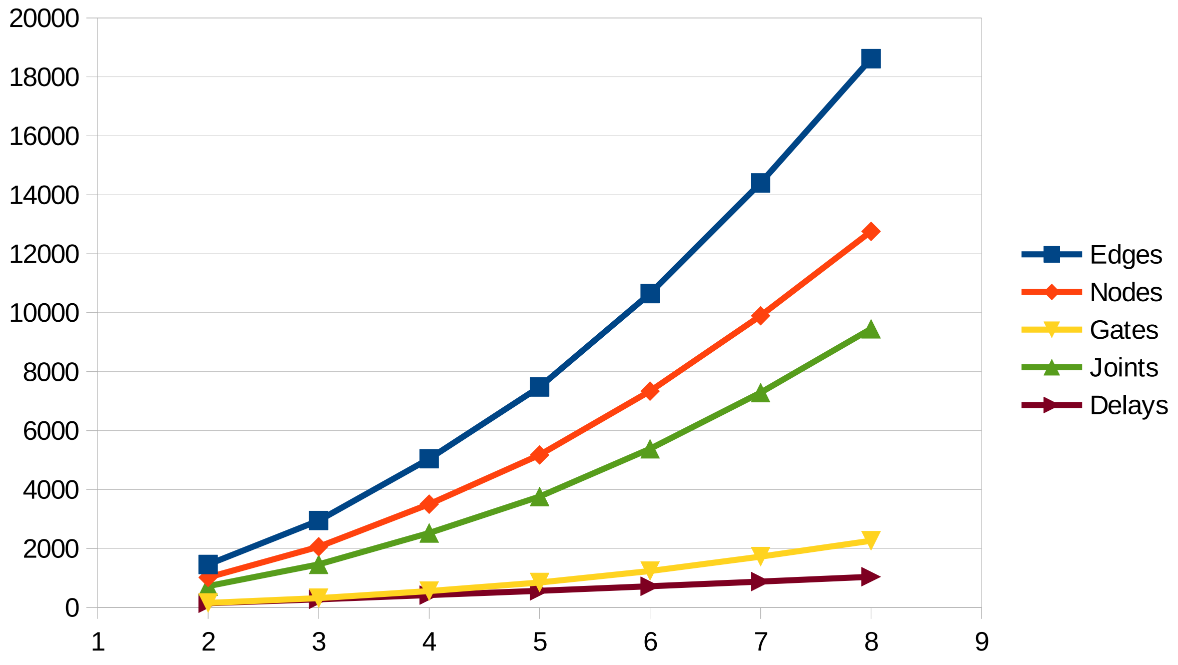
```
iter(base on 3)
       iter(dest on 3)
              {
              aux user(base, dest);

              user(base, dest) --> one(hint(k / 3, k % 3), k is 3 * base + dest, k on 9);
              }
```

hanoi.c client interface



hanoi.tin complexity law



Complexity

In temporal inference networks, recursions on the set of timeless logic propositions are expanded in space by the compiler, whereas recursions in time are delayed up to the time of their evaluation by the inference engine, hence these networks grow in size with the number of disks n and the corresponding complexity law has been studied. For this system, it appears to be $O(n^3)$ for both edges and nodes, but the number of steps needed to reach the solution is instead $O(2^n)$ and so the temporal horizon. It has been proven[3] that the algorithm implemented by the inference engine runs in real time if the discrete time unit is chosen above a threshold, which grows with the product of the network size for the maximum time displacement represented in the specification and is thus independent from the horizon, but in this case the second factor of the product unfortunately follows the same exponential law of the horizon for the presence of the drive. Once removed the drive, the polynomial law is actually followed by the threshold.

Conclusions

Despite its weaknesses, deterministic reasoning can sometimes solve constraint satisfaction problems if the problems themselves are described in a proper way. Temporal logic help to separate the structure of a dynamical system from its state, opening to the possibility of an execution on an unlimited temporal horizon. TINX Suite provides some interesting tools to validate system specifications and also to execute them in real time, if their description is mandatory enough to allow a deterministic resolution.

3 A. Giotti (2000) in the thesis available on ResearchGate at the link: “https://www.researchgate.net/publication/334317800_Un_Esecutore_di_Specifiche_Logiche_in_Tempo_Reale”