UNIVERSITEIT GENT

**Formele Logica** Academiejaar 2014-2015

# Project Alloy

**Jasper D'haene Florian Dejonckheere**

# 1 Ramsey-getallen

Ramsey's theorem dicteert dat in een voldoende grote set waarvan de bogen gekleurd zijn met een willekeurig aantal kleuren, monochromatisch gekleurde subsets te vinden zijn. Stel $m, n \in \mathbb{R}^+$ en twee kleuren $k_i, i \in [0, 1]$, dan definieert Ramsey's theorem $R(m, n, k)$ de ondergrens voor de complete graaf die een subset over tenminste $m$ toppen met kleur $k_0$, of een subset over tenminste $n$ toppen met kleur $k_1$ bevat.

Informeel is dit probleem (voor $k = 2$) ook wel bekend als het *party problem*: hoe groot is de minimale set van personen die uitgenodigd moeten worden voor een feestje, waarvoor geldt dat ofwel minstens $m$ personen elkaar (mutueel) kennen, ofwel $n$ personen elkaar (mutueel) niet kennen.

Listing 1: Ramseygetallen Alloy model

```
/**
 * ramsey_numbers.als - Calculating Ramsey numbers in Alloy 4
 *
 * Florian Dejonckheere <florian@floriandejonckheere.be>
 * Jasper D'haene <jasper.dhaene@gmail.com>
 *
 * */

/**
 * Om dit probleem wat interessanter en modelleerbaarder
 * te maken, stellen we dat de te calculeren Ramseyget∀en
 * gewoon het antwoord zijn van het 'party problem':
 *
 * R(m,n) = Hoeveel gasten moeten er uitgenodigd worden
 * zodat ≥ m elkaar kennen en ≥ n elkaar niet kennen.
 *
 * Hint: R(3, 3) = 6 [1]
 * Hint: R(3, 3) = R(3, 3, 2)
 * Hint: R(3) = R(3, 3, 3) = R(3, 3, 3; 3)
 * Hint: R(3, 3, 3) = 17 [2]
 *
 * [1] http://mathworld.wolfram.com/RamseyNumber.html
 * [2] http://en.wikibooks.org/wiki/Combinatorics/Bounds_for_Ramsey_numbers
 *
 * */

sig Colour {}
sig Node {}

sig Edge {
        connection: Node -> Node,
        colour: one Colour
}{
        // No self-referencing
        ∀ node:Node • (node -> node not in connection)
}

// Make sure that 'colouring' is the same as Edge->colour
fact {
        colour = ~(Graph.colouring)
}

// Specify the colour conditions
fact {
        // There are X edges in the same colour and Y in a different. X+Y=#Edge. X and Y are even.
        some col:Colour • #((~(Graph.colouring)).col) = 4
        some col:Colour • #((~(Graph.colouring)).col) = 2
}


one sig Graph{
        nodes: set Node,

        // Edges: Node -> Node
        edges: set Edge,
        colouring: Colour one -> some Edge
}{
        // All nodes in graph
        ∀ node: Node • node in nodes

        // All edges in graph
        ∀ edge:Edge • edge in edges

        // Edges relationship is symmetrical
        ∀ edge: Edge • some edge':Edge • edge.connection = ~(edge'.connection)

        // Every edge only connects 2 points
        ∀ edge: Edge • one edge.connection

        // Complete graph
        ∀ n, n' : Node • some e:Edge • n ≠ n' ⟹ n -> n' in e.connection
}
```

```
run {} for 1 Graph, exactly 2 Colour, exactly 3 Node, exactly 6 Edge
```

## 2 Cyclische toren van Hanoi

De cyclische toren van Hanoi is een variant op de bekende combinatorische puzzel. Buiten de drie hoofdregels geldt de volgende regels ook:

4. Schijven kunnen enkel cyclisch naar rechts opgeschoven worden. Voor drie palen $A, B, C$ is dit dus $A \to B \to C \to A$

Als basis voor het algoritme werd de geïmplementeerde versie van Ilya Shlyakhter beschouwd. Deze versie wordt bijgeleverd als voorbeeld bij de Alloy Analyzer, maar implementeert de niet-cyclische variant.

Listing 2: Cyclische toren van Hanoi Alloy model

```
module examples/puzzles/hanoi

/*
 * Cyclic towers of Hanoi model
 *
 * author of hanoi model: Ilya Shlyakhter
 * modified by Jasper D'haene <jasper.dhaene@gmail.com>
 */

open util/ordering[State] as states
open util/ordering[Stake] as stakes
open util/ordering[Disc] as discs

sig Stake { }

sig Disc { }

/**
 * sig State: the complete state of the system --
 * which disc is on which stake.  An solution is a
 * sequence of states.
 */
sig State {
  on: Disc -> one Stake  // _each_ disc is on _exactly one_ stake
  // note that we simply record the set of discs on each stake --
  // the implicit assumption is that on each stake the discs
  // on that stake are ordered by size with sm∀est disc on top
  // and largest on bottom, as the problem requires.
}

/**
 * compute the set of discs on the given stake in this state.
 * ~(this.on) map the stake to the set of discs on that stake.
 */
fun discsOnStake[st: State, stake: Stake]: set Disc {
  stake.~(st.on)
}

/**
 * compute the top disc on the given stake, or the empty set
 * if the stake is empty
 */
fun topDisc[st: State, stake: Stake]: lone Disc {
  { d: st.discsOnStake[stake] • st.discsOnStake[stake] in discs/nexts[d] + d }
}

/**
 * Describes the operation of moving the top disc from stake fromStake
 * to stake toStake.  This function is defined implicitly but is
 * nevertheless deterministic, i.e. the result state is completely
 * determined by the initial state and fromStake and toStake; hence
 * the "det" modifier above.  (It's important to use the "det" modifier
 * to tell the Alloy Analyzer that the function is in fact deterministic.)
 */
pred Move [st: State, fromStake, toStake: Stake, s': State] {
   //modified: Cyclic move property
```

```
   fromStake ≠stakes/last ⟹
      fromStake.next = toStake
   else
      toStake = stakes/first

   let d = st.topDisc[fromStake] • {
      // ∀ discs on toStake must be larger than d,
      // so that we can put d on top of them
      st.discsOnStake[toStake] in discs/nexts[d]
      // after, the fromStake has the discs it had before, minus d
      s'.discsOnStake[fromStake] = st.discsOnStake[fromStake] - d
      // after, the toStake has the discs it had before, plus d
      s'.discsOnStake[toStake] = st.discsOnStake[toStake] + d
      // the remaining stake afterwards has exactly the discs it had before
      let otherStake = Stake - fromStake - toStake •
        s'.discsOnStake[otherStake] = st.discsOnStake[otherStake]
   }
}

/**
 * there is a leftStake that has ∀ the discs at the beginning,
 * and a rightStake that has ∀ the discs at the end
 */
pred Game1 {
   Disc in states/first.discsOnStake[stakes/first]
   //modified end condition
   some finalState: State • Disc in finalState.discsOnStake[stakes/last] ∧ (finalState = states/last)

   // each adjacent pair of states are related by a valid move of one disc
   ∀ preState: State - states/last •
       let postState = states/next[preState] •
          some fromStake: Stake • {
             // must have at least one disk on fromStake to be able to move
             // a disc from fromStake to toStake
             some preState.discsOnStake[fromStake]
             // post- results from pre- by making one disc move
             some toStake: Stake • preState.Move[fromStake, toStake, postState]
          }
}

/**
 * there is a leftStake that has ∀ the discs at the beginning,
 * and a rightStake that has ∀ the discs at the end
 */
pred Game2  {
   Disc in states/first.discsOnStake[stakes/first]
   some finalState: State • Disc in finalState.discsOnStake[stakes/last]

   // each adjacent pair of states are related by a valid move of one disc
   ∀ preState: State - states/last •
       let postState = states/next[preState] •
          some fromStake: Stake •
             let d = preState.topDisc[fromStake] • {
               // must have at least one disk on fromStake to be able to move
               // a disc from fromStake to toStake
               some preState.discsOnStake[fromStake]
               postState.discsOnStake[fromStake] = preState.discsOnStake[fromStake] - d
               some toStake: Stake • {
                 // post- results from pre- by making one disc move
                 preState.discsOnStake[toStake] in discs/nexts[d]
                 postState.discsOnStake[toStake] = preState.discsOnStake[toStake] + d
                // the remaining stake afterwards has exactly the discs it had before
                let otherStake = Stake - fromStake - toStake •
                    postState.discsOnStake[otherStake] = preState.discsOnStake[otherStake]
                }
             }
       }

//#state == 22 + 3X voor geldige instantie.
run Game1 for 1 but 3 Stake, 3 Disc, 22 State expect 1
run Game2 for 1 but 3 Stake, 3 Disc, 8 State expect 1
```