LOGICAL MODELS OF PROBLEMS AND COMPUTATIONS

Case Studies

Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

JKU
JOHANNES KEPLER
UNIVERSITY LINZ

1. **Greatest Common Divisor**
2. Propositional Satisfiability
3. N Queens
4. Crab and Racoon
5. Goat, Wolf, and Cabbage
6. Knight’s Tour
7. Tower of Hanoi
8. Hotel Room Locking
9. A Robotic Controler
10. Steam Boiler Control

Greatest Common Divisor

Problem: given m and n, compute the greatest common divisor (gcd) g of m and n.

```
fun gcd(m:nat,n:nat): nat
  requires hasgcd(m,n);
  ensures isgcd(result,m,n);
= ...
```

- What is the domain of m, n, g?
- What is a “divisor”?
- What is a “common” divisor?
- What is “the greatest” such common divisor?
- Does such a gcd always exist?
- Does not more than one such a gcd exist?

Greatest Common Divisor

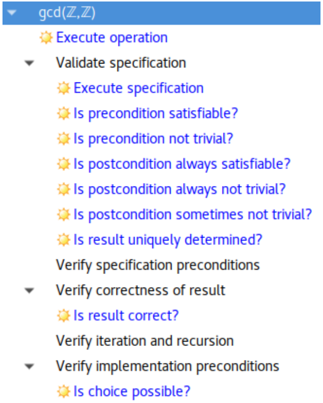
```
val N: ℕ;        // the domain size
type nat = ℕ[N]; // the domain itself

// when do m and n have a gcd?
pred hasgcd(m:nat,n:nat) ⇔ m ≠ 0 ∨ n ≠ 0;

// the predicate "m divides n"
pred divides(m:nat,n:nat) ⇔ ∃p:nat. m·p = n;

// the predicate "g is a gcd of m and n"
pred isgcd(g:nat,m:nat,n:nat) ⇔
  // g is a common divisor of m and n
  divides(g,m) ∧ divides(g,n) ∧
  // g is the greatest such value
  ¬∃g0:nat. divides(g0,m) ∧ divides(g0,n) ∧ g0 > g;
```

Checking Theorems

```
// the gcd exists and is unique
theorem gcdExists(m:nat,n:nat) ⇔
  hasgcd(m,n) ⇒ ∃g:nat. isgcd(g,m,n);
theorem gcdUnique(m:nat,n:nat) ⇔
  hasgcd(m,n) ⇒ ∀g1:nat,g2:nat. isgcd(g1,m,n) ∧ isgcd(g2,m,n) ⇒ g1 = g2;
```

```
Using N=20.
Type checking and translation completed.
Executing gcdExists(ℤ,ℤ) with all 441 inputs.
Execution completed for ALL inputs (170 ms, 441 checked, 0 inadmissible).
Executing gcdUnique(ℤ,ℤ) with all 441 inputs.
305 inputs (305 checked, 0 inadmissible, 0 ignored)...
Execution completed for ALL inputs (2731 ms, 441 checked, 0 inadmissible).
```

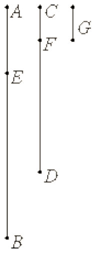
Validating the Specification


gcd(ℤ,ℤ)
Execute operation
Validate specification
Execute specification
Is precondition satisfiable?
Is precondition not trivial?
Is postcondition always satisfiable?
Is postcondition always not trivial?
Is postcondition sometimes not trivial?
Is result uniquely determined?
Verify specification preconditions
Verify correctness of result
Is result correct?
Verify iteration and recursion
Verify implementation preconditions
Is choice possible?

```
fun gcd(m:nat,n:nat): nat
  requires hasgcd(m,n);
= choose g:nat with isgcd(g,m,n);
```

```
Using N=20.
Type checking and translation completed.
Executing gcd(ℤ,ℤ) with all 441 inputs.
Execution completed for ALL inputs (302 ms, 440 checked, 1 inadmi
Executing _gcd_2_Spec(ℤ,ℤ) with all 441 inputs.
Execution completed for ALL inputs (201 ms, 440 checked, 1 inadmi
Executing _gcd_2_PreSat().
Execution completed (0 ms).
Executing _gcd_2_PreNotTrivial().
Execution completed (0 ms).
...
Executing _gcd_2_PreOp0(ℤ,ℤ) with all 441 inputs.
Execution completed for ALL inputs (149 ms, 440 checked, 1 inadmissible).
```

Euclid’s Algorithm



Let AB and CD be two given numbers not relatively prime. It is required to find the greatest common measure of AB and CD.

If now CD measures AB, since it also measures itself, then CD is a common measure of CD and AB. And it is clear that it is also the greatest, for no greater number than CD measures CD.

But, if CD does not measure AB, then, when the less of the numbers AB and CD being continually subtracted from the greater, some number is left which measures the one before it. For a unit is not left, otherwise AB and CD would be relatively prime, which is contrary to the hypothesis. Therefore some number is left which measures the one before it.

Now let CD, measuring BE, leave EA less than itself, let EA, measuring DF, leave FC less than itself, and let CF measure AE.

Since then, CF measures AE, and AE measures DF, therefore CF also measures DF. But it measures itself, therefore it also measures the whole CD.

But CD measures BE, therefore CF also measures BE. And it also measures EA, therefore it measures the whole BA.

But it also measures CD, therefore CF measures AB and CD. Therefore CF is a common measure of AB and CD.

. . .

Euclid’s Algorithm (Continued)

. . .

I say next that it is also the greatest.

If CF is not the greatest common measure of AB and CD, then some number G, which is greater than CF, measures the numbers AB and CD.

Now, since G measures CD, and CD measures BE, therefore G also measures BE. But it also measures the whole BA, therefore it measures the remainder AE.

But AE measures DF, therefore G also measures DF. And it measures the whole DC, therefore it also measures the remainder CF, that is, the greater measures the less, which is impossible.

Therefore no number which is greater than CF measures the numbers AB and CD. Therefore CF is the greatest common measure of AB and CD. Q.E.D.

Euclid’s Elements, Book VII, Proposition 2:
“To find the greatest common measure of two given numbers not relatively prime.”

The Knowledge in Euclid’s Algorithm

```
// the essential knowledge on which Euclid's algorithm is based:

// original input condition:
//   "Let AB and CD be two given numbers not relatively prime."
// we are more general and allow them to be relatively prime (have gcd one)
// but we still require that not both lengths may be zero
// (Ancient Greeks did not consider "0" as appropriate lengths/numbers)

// the original termination criterion:
//   "If now CD measures AB, since it also measures itself,
//    then CD is a common measure of CD and AB."
// however, rather than stating divides(m,n) ⇒ gcd(m,n) = m
// we let the algorithm run one more step to end up with 0
// which gives an easier termination criterium
theorem gcd0(m:nat) ⇔ m≠0 ⇒ gcd(m,0) = m;

...
```

The Knowledge in Euclid’s Algorithm (Continued)

```
...

// order apparently does not matter:
//   "... when the less of the numbers AB and CD being
//   continually subtracted from the greater ..."
theorem gcd1(m:nat,n:nat) ⇔ m ≠ 0 ∨ n ≠ 0 ⇒ gcd(m,n) = gcd(n,m);

// the core idea:
//   "But, if CD does not measure AB, then, when the less of the numbers
//   AB and CD being continually subtracted from the greater,
//   some number is left which measures the one before it."
// here "continuous subtraction" is the same as "remainder computation"
theorem gcd2(m:nat,n:nat) ⇔ 1 ≤ n ∧ n ≤ m ⇒ gcd(m,n) = gcd(m%n,n);
```

Euclid’s Algorithm as a Function

```
theorem gcd0(m:nat) ⇔ m≠0 ⇒ gcd(m,0) = m;
theorem gcd1(m:nat,n:nat) ⇔ m ≠ 0 ∨ n ≠ 0 ⇒ gcd(m,n) = gcd(n,m);
theorem gcd2(m:nat,n:nat) ⇔ 1 ≤ n ∧ n ≤ m ⇒ gcd(m,n) = gcd(m%n,n);

fun gcdf(m:nat,n:nat): nat
  requires hasgcd(m,n);      // the precondition
  ensures isgcd(result,m,n); // the postcondition
  decreases m+n;             // the termination measure
= if m = 0 then n
  else if n = 0 then m
  else if m > n then gcdf(m%n,n)
  else gcdf(m,n%m);
```

Euclid’s Algorithm as a Procedure

```
proc gcdp(m:nat,n:nat): nat
  requires hasgcd(m,n);      // the precondition
  ensures isgcd(result,m,n); // the postcondition
{
  var a:nat := m;
  var b:nat := n;
  while a > 0 ∧ b > 0 do
    invariant hasgcd(a,b);                          // a loop invariant
    invariant ∀r:nat. isgcd(r,a,b) ⇔ isgcd(r,m,n); // a loop invariant
    decreases a+b;                                  // the termination measure
  {
    if a > b then
      a := a%b;
    else
      b := b%a;
  }
  return if a = 0 then b else a;
}
```

Euclid’s Algorithm as a Transition System

```
// initial state condition and next state relation
pred init(a:nat,b:nat) ⇔ hasgcd(a,b);
pred next(a:nat,b:nat,a0:nat,b0:nat) ⇔
  if a > b
    then a0 = a%b ∧ b0 = b
    else a0 = a ∧ b0 = b%a;

proc gcds(m:nat,n:nat): nat
  requires init(m,n);
  ensures isgcd(result,m,n);
{
  var a:nat = m; var b:nat = n;
  while a > 0 ∧ b > 0 do
  {
    choose a0:nat,b0:nat with next(a,b,a0,b0);
    a := a0; b := b0;
  }
  return if a = 0 then b else a;
}
```

1. Greatest Common Divisor
2. **Propositional Satisfiability**
3. N Queens
4. Crab and Racoon
5. Goat, Wolf, and Cabbage
6. Knight’s Tour
7. Tower of Hanoi
8. Hotel Room Locking
9. A Robotic Controler
10. Steam Boiler Control

Propositional Satisfiability

The SAT Problem: given a propositional formula f in conjunctive normal form, decide whether f is satisfiable (i.e., whether there exists an assignment of truth values to the boolean variables in f that satisfies f, i.e., that makes f true).

```
pred DPLL(f:Formula)
  ensures result ⇔ satisfiable(f);
⇔ ...
```

- Modeling the (syntactic) domain of “propositional formulas” in “conjunctive normal form”.
- Modeling the (semantic) notions of “assignments”, “satisfaction”, and “satisfiability”.

Formulas in Conjunctive Normal Form

Formulas such as $(\neg x \vee y \vee \neg z) \wedge (y \vee z) \wedge (x \vee \neg y \vee \neg z)$ consisting of “clauses” like $(x \vee \neg y \vee \neg z)$ which consist of (positive or negative) “literals” like x or $\neg y$.

```
// the number of literals
val N: ℕ;

// the types
type Literal = ℤ[-N,N] with value ≠ 0;
type Clause  = Set[Literal] with ¬∃l∈value. -l∈value;
type Formula = Set[Clause];
type Valuation  = Set[Literal] with ¬∃l∈value. -l∈value;
```

Satisfiability

```
// the satisfaction relation
pred satisfies(v:Valuation, l:Literal) ⇔ l∈v;
pred satisfies(v:Valuation, c:Clause) ⇔ ∃l∈c. satisfies(v, l);
pred satisfies(v:Valuation, f:Formula) ⇔ ∀c∈f. satisfies(v,c);

// the satisfiability of a formula
pred satisfiable(f:Formula) ⇔ ∃v:Valuation. satisfies(v,f);
```

Satisfiability versus Validity

```
// the validity of a formula
pred valid(f:Formula) ⇔ ∀v:Valuation. satisfies(v,f);

// the negation of a formula
fun not(f: Formula):Formula =
  { c | c:Clause with ∀d∈f. ∃l∈d. -l∈c };

// the duality of satisfiability and validity
theorem validIsNotSatNeg(f:Formula) ⇔
  valid(f) ⇔ ¬satisfiable(not(f));
```

A Simple SAT Solver

```
// the literals of a formula
fun literals(f:Formula):Set[Literal] = {l | l:Literal with ∃c∈f. l∈c};

// the result of setting a literal l in formula f to true
fun substitute(f:Formula,l:Literal):Formula = {c\{-l} | c∈f with ¬(l∈c)};

// the recursive DPLL algorithm (without optimizations)
multiple pred DPLL(f:Formula)
  ensures result ⇔ satisfiable(f);
  decreases |literals(f)|;
⇔
  if f = ∅[Clause] then
    ⊤
  else if ∅[Literal] ∈ f then
    ⊥
  else
    choose l∈literals(f) in
    DPLL(substitute(f,l)) ∨ DPLL(substitute(f,-l));
```

1. Greatest Common Divisor

2. Propositional Satisfiability

3. **N Queens**

4. Crab and Racoon

5. Goat, Wolf, and Cabbage

6. Knight’s Tour

7. Tower of Hanoi

8. Hotel Room Locking

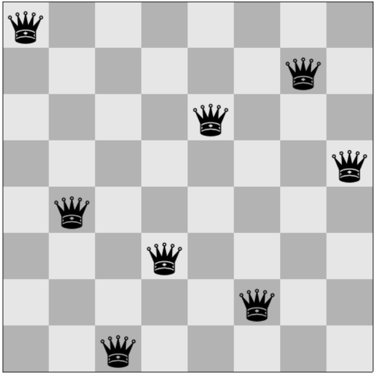
9. A Robotic Controler

10. Steam Boiler Control

N Queens

Problem: given a chess board of dimension N, place N queens on the board such that they do not beat each other.

```
fun queens(n:Num):Queens
  requires queensExist(n);
= choose q:Queens with queens(n, q);
```



blogs.mathworks.com

The Problem Specification

```
// the size of the board
val N:ℕ;
axiom notZero ⇔ N ≠ 0;
type Num = ℕ[N];
type Index = ℕ[N-1];
type Queens = Array[N,Index];
...

// an arbitrary solution to the N-queen problem
fun queens(): Queens
  requires queensExist(N);
= queens(N);

// all solutions to the n-queen respectively N-queens problem
fun queensAll(n:Num): Set[Queens] = { q | q:Queens with queens(n, q) };
fun queensAll(): Set[Queens] = queensAll(N);
```

The Postcondition

```
// a necessary and sufficient condition for
// the existence of a solution to the n-queens problem
pred queensExist(n:Num) ⇔ n ≠ 2 ∧ n ≠ 3;
theorem queensExistValid(n:Num) ⇔ queensExist(n) ⇔ ∃q:Queens. queens(n,q);

// do queens at (i1,j1) and (i2,j2) beat each other?
pred beats(i1:Index, j1:Index, i2:Index, j2:Index)
⇔
  i1 = i2 ∨ j1 = j2 ∨
  i1-j1 = i2-j2 ∨
  i1+j1 = i2+j2;

// q is a solution to the n-queens problem
pred queens(n:Num, q:Queens) ⇔
  (∀k:Index with k < n. q[k] < n) ∧ (∀k:Index with n ≤ k. q[k] = 0) ∧
  ¬(∃k1:Index, k2:Index with k1 < k2 ∧ k2 < n. beats(k1, q[k1], k2, q[k2]));
```

Solving the Problem

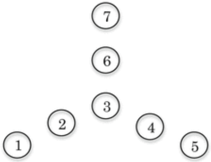
```
Using N=7.
Computing the truth value of notZero...
Type checking and translation completed.
Executing queens().
Branch 0 of nondeterministic function queens():
Result (518 ms): [5,3,1,6,4,2,0]
...
Result (205 ms): [1,3,5,0,2,4,6]
Branch 40 of nondeterministic function queens():
No more results (8823 ms).
Execution completed (8825 ms).
Executing queensAll().
Run of deterministic function queensAll():
Result (8427 ms): {[5,3,1,6,4,2,0],[4,1,5,2,6,3,0],[3,6,2,5,1,4,0],[2,4,6,1,3,5,0],
[5,2,4,6,0,3,1],[6,4,2,0,5,3,1],[5,2,6,3,0,4,1],[5,3,6,0,2,4,1],[5,2,0,3,6,4,1],
...
[1,4,2,0,6,3,5],[4,2,0,5,3,1,6],[3,0,4,1,5,2,6],[2,5,1,4,0,3,6],[1,3,5,0,2,4,6]}
Execution completed (8430 ms).
```

1. Greatest Common Divisor
2. Propositional Satisfiability
3. N Queens
4. **Crab and Racoon**
5. Goat, Wolf, and Cabbage
6. Knight’s Tour
7. Tower of Hanoi
8. Hotel Room Locking
9. A Robotic Controler
10. Steam Boiler Control

Crab and Racoon

Problem: A crafty crab has constructed seven blind holes in the configuration shown. Every day at noon it changes from one hole to an adjacent hole, and every night at midnight a raccoon comes and looks in a single hole in the hopes of finding the crab. After five consecutive days of failure, the raccoon takes a day off in an attempt to discover a hole-checking protocol that will guarantee it will catch the crab. In other words, the raccoon is going to do some System 2 thinking to discover a sequence of hole-checking that will guarantee it catches the crab even if the crab knows the hole-checking protocol. What sequence of hole-checking will inevitably trap the crab?

Problem 18.1 from E.F.Meyer III et al., *Guide to Teaching Puzzle-based Learning*, 2014.



```
// solve problem for fixed crab burrow
fun solve(): Holes = choose looks:Holes with solution(looks);
```

Modeling the Domain

```
// the burrow as a directed graph with node set and edge relation
type Hole = ℕ[6];
val Move = {
  ⟨0,1⟩, ⟨1,0⟩, ⟨1,2⟩, ⟨2,1⟩, ⟨2,3⟩, ⟨2,5⟩,
  ⟨3,2⟩, ⟨3,4⟩, ⟨4,3⟩, ⟨5,2⟩, ⟨5,6⟩, ⟨6,5⟩
};
pred move(i:Hole,j:Hole) ⇔ ∃m∈Move. (m.1 = i ∧ m.2 = j);

// maximum number of looks considered
val N:ℕ;
type Num = ℕ[N];
type Holes = Array[N,Hole];

// m describes a possible sequence of N moves of the crab
pred movement(m:Holes) ⇔ ∀i:Num with i < N-1. move(m[i],m[i+1]);

// the set of all moves
val Moves = { m | m:Holes with movement(m) };
```

Modeling the Problem

```
// sequence "looks" catches crab moving along sequence "moves"
pred catch(looks:Holes, moves:Holes) ⇔ ∃i:Num with i < N. looks[i] = moves[i];

// sequence "looks" is guaranteed to catch the crab
pred solution(looks:Holes) ⇔ ∀moves∈Moves. catch(looks, moves);

// compute a solution of the problem
fun solve(): Holes = choose looks:Holes with solution(looks);

// also a violation of this theorem determines a solution
// (checking this allows the application of multiple threads to each candidate)
theorem nosolution(looks:Holes) ⇔ ¬solution(looks);
```

Auxiliary Operations

```
// to speed up the computation, we compute the set of possible moves constructively

// compute the set of all sequences of n moves of the crab
fun moves(n:Num):Set[Holes]
  requires n ≠ 0;
= if n = 1 then
    { Array[N,Hole](i) | i:Hole }
  else let M = moves(n-1) in
    { m with [n-1]=i | m ∈ M, i:Hole with move(m[n-2],i) };

// the set of all sequences of N moves of the crab
val Moves = moves(N);

// we only have correct moves
theorem movesCorrect ⇔ ∀m∈Moves. movement(m);
```

Solving a Simpler Problem

```
// holes 0 - 1 - 2 - 3 - 4
type Hole = ℕ[4];
val Move = {
  ⟨0,1⟩, ⟨1,0⟩, ⟨1, 2⟩, ⟨2,1⟩, ⟨2,3⟩, ⟨3,2⟩, ⟨3,4⟩, ⟨4,3⟩
};
```

```
Using N=10.
Computing the value of Moves...
Computing the truth value of movesCorrect...
Computing the value of looks...
Type checking and translation completed.
Executing nosolution(Array[ℤ]) with all 9765625 inputs.
PARALLEL execution with 4 threads (output disabled).
ERROR in execution of nosolution([3,2,1,3,2,1,0,0,0,0]): evaluation of
  nosolution
at line 96 in file c.txt:
  theorem is not true
ERROR encountered in execution.
```

Solving the Real Problem

```
Using N=10.
Computing the value of Moves...
Computing the truth value of movesCorrect...
Computing the value of looks...
Type checking and translation completed.
Executing nosolution(Array[ℤ]) with all 282475249 inputs.
PARALLEL execution with 4 threads (output disabled).
28037 inputs (20826 checked, 0 inadmissible, 0 ignored, 7211 open)...
46471 inputs (38524 checked, 0 inadmissible, 0 ignored, 7947 open)...
...
54665655 inputs (54663958 checked, 0 inadmissible, 0 ignored, 1697 open)...
54679254 inputs (54675189 checked, 0 inadmissible, 0 ignored, 4065 open)...
ERROR in execution of nosolution([5,2,3,2,1,5,2,3,2,1]): evaluation of
  nosolution
at line 96 in file crab.txt:
  theorem is not true
ERROR encountered in execution.
```

1. Greatest Common Divisor
2. Propositional Satisfiability
3. N Queens
4. Crab and Racoon
5. **Goat, Wolf, and Cabbage**
6. Knight’s Tour
7. Tower of Hanoi
8. Hotel Room Locking
9. A Robotic Controler
10. Steam Boiler Control

Goat, Wolf, and Cabbage

Problem: a farmer has to cross a river with a goat, a wolf, and a cabbage. The only boat can carry apart from the farmer only another passenger. How can the farmer cross the river such that the wolf does not eat the goat or the goat does not eat the cabbage?

Problem 18 of *propositiones ad acuendos iuvenes (problems to sharpen the young)*, 9th century.

Managing Planning Problems by Logical Solving

- Translation of problem into a transition system.
  - Initial situation $\rightsquigarrow$ initial state condition $I(s)$
  - Actions $\rightsquigarrow$ transition relation $R(s, s')$
  - Final situation $\rightsquigarrow$ termination condition $T(s)$
- Check satisfiability of the following formulas
  1. $F_0 \equiv I(s_0) \wedge T(s_0)$
  2. $F_1 \equiv I(s_0) \wedge R(s_0, s_1) \wedge T(s_1)$
  3. $F_2 \equiv I(s_0) \wedge R(s_0, s_1) \wedge R(s_1, s_2) \wedge T(s_2)$
  4. $\ldots$
  - F_k: in k steps the desired target situation is reached.
  - If F_k is satisfiable, the satisfying assignments for $s_0, s_1, \ldots, s_k$ represent the sequence of situations to reach the goal.
- If there are only n possible situations but $F_0, \ldots, F_{n-1}$ are not satisfiable, then there is no possible plan.
  - In practice, human chooses maximum value for k.

Modeling the Domain

```
// the maximum number of moves considered
val N:ℕ;
type Num = ℕ[N];

// the entities to be transfered across the river
// enumtype Entity = farmer | goat | wolf | cabbage;
type Entity = ℕ[3];
val farmer = 0; val goat = 1; val wolf = 2; val cabbage = 3;

// the sides of the river
// enumtype Place = left | right;
type Place = Bool; val left  = false; val right = true;

// the side opposite to p
fun other(p:Place):Place = ¬p;

// a placement of the entity to river sides
// and a sequence of such placements (including the initial one)
type Placement = Map[Entity,Place];
type PlacementSeq = Array[N+1,Placement];
```

Modeling the Transition System

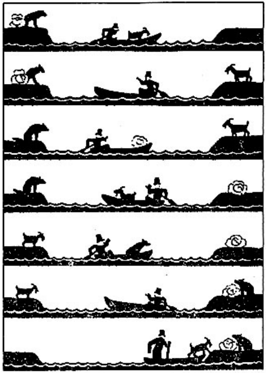
```
// the initial and the goal situation
pred init(p:Placement) ⇔ ∀e:Entity. p[e] = left;
pred goal(p:Placement) ⇔ ∀e:Entity. p[e] = right;

// placement p is safe
pred safe(p:Placement) ⇔
  let q = other(p[farmer]) in ¬(p[wolf] = q ∧ p[goat] = q) ∧ ¬(p[goat] = q ∧ p[cabbage] = q);

// from placement p we derive q by a transfer of the farmer alone
pred move0(p:Placement, q:Placement) ⇔
  p[farmer] ≠ q[farmer] ∧ ∀e:Entity with e ≠ farmer. p[e] = q[e];

// from p we get q by a transfer of farmer together with entity e
pred move1(p:Placement, q:Placement, e:Entity) ⇔
  p[farmer] ≠ q[farmer] ∧ p[e] = p[farmer] ∧ q[e] = q[farmer] ∧
  ∀e0:Entity with e0 ≠ farmer ∧ e0 ≠ e. p[e0] = q[e0];

// from placement p we derive a safe placement q by some transfer
pred move(p:Placement, q:Placement) ⇔
  safe(q) ∧ (move0(p, q) ∨ (∃e:Entity with e ≠ farmer. move1(p, q, e)));
```

Modeling the System Execution

```
proc Farmer(p0:Placement): Tuple[Bool,Num,PlacementSeq]
  requires init(p0);
{
  var ps: PlacementSeq = Array[N+1,Placement](p0);
  var p:Placement = p0;
  assert safe(p);
  var i:Num = 0;
  while i < N ∧ ¬goal(p) do
  {
    choose p1:Placement with move(p, p1);
    p := p1;
    assert safe(p);
    ps[i+1] := p;
    i := i+1;
  }
  return ⟨goal(p),i,ps⟩;
}

theorem noSolution() ⇔ ∀p:Placement with init(p).
  let r = Farmer(p) in if r.1 then print ⟨r.2,r.3⟩ in false else true;
```

Computing the Solution

```
Using N=7.
Computing the value of farmer...
Computing the value of goat...
Computing the value of wolf...
Computing the value of cabbage...
Computing the value of left...
Computing the value of right...
Type checking and translation completed.
Executing noSolution().
[7,[[false,false,false,false],[true,true,false,false],
  [false,true,false,false],[true,true,true,false],
  [false,false,true,false],[true,false,true,true],
  [false,false,true,true],[true,true,true,true]]]
ERROR in execution of noSolution(): evaluation of
  noSolution
at line 92 in file goat.txt:
  theorem is not true
ERROR encountered in execution.
```



http://www.hirnwindungen.de

Alternative: An Action-Oriented Model

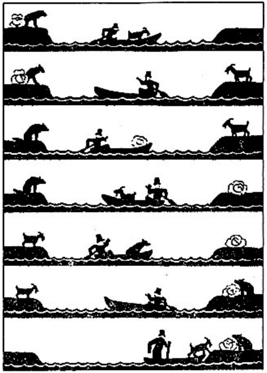
```
// a transfer of the farmer alone or with an entity
rectype(1) Action = farmer1 | farmer2(Entity);
type ActionSeq = Array[N,Action];

// perform move denoted by action a in current placement p and return new placement
fun move(a:Action, p:Placement): Placement =
  let p0 = other(p[farmer]) in
  match a with
  {
    farmer1 -> p with [farmer] = p0;
    farmer2(e:Entity) -> p with [farmer] = p0 with [e] = p0;
  };

pred admissible(a:Action, p:Placement) ⇔ // action a is admissible in current placement p
  match a with
  {
    farmer1 -> ⊤;
    farmer2(e:Entity) -> p[e] = p[farmer];
  } ∧ safe(move(a,p));
```

Alternative: An Action-Oriented Model

```
proc Farmer0(p0:Placement): Tuple[Bool,Num,ActionSeq]
  requires init(p0);
{
  var p:Placement = p0;
  assert safe(p);
  var as:ActionSeq = Array[N,Action](Action!farmer1);
  var i:Num = 0;
  while i < N ∧ ¬goal(p) do
  {
    choose a:Action with admissible(a, p);
    p := move(a,p);
    assert safe(p);
    as[i] := a;
    i := i+1;
  }
  return ⟨goal(p),i,as⟩;
}

theorem noSolution0() ⇔ ∀p:Placement with init(p).
  let r = Farmer0(p) in if r.1 then print ⟨r.2,r.3⟩ in false else true;
```

Alternative: An Action-Oriented Model

```
Using N=7.
Computing the value of farmer...
Computing the value of goat...
Computing the value of wolf...
Computing the value of cabbage...
Computing the value of left...
Computing the value of right...
Type checking and translation completed.
Executing noSolution0().
[7,[0:1:0:[1],0:0:0:[],0:1:0:[2],
    0:1:0:[1],0:1:0:[3],0:0:0:[],
    0:1:0:[1]]]
ERROR in execution of noSolution0(): evaluation of
  noSolution0
at line 142 in file goat.txt:
  theorem is not true
ERROR encountered in execution.
```

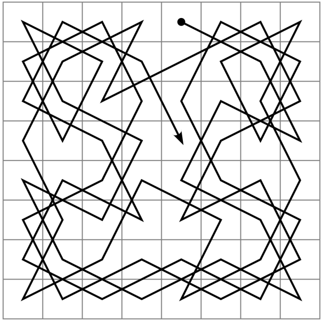


http://www.hirnwindungen.de

1. Greatest Common Divisor
2. Propositional Satisfiability
3. N Queens
4. Crab and Racoon
5. Goat, Wolf, and Cabbage
6. **Knight's Tour**
7. Tower of Hanoi
8. Hotel Room Locking
9. A Robotic Controler
10. Steam Boiler Control

Knight’s Tour

Problem: given a chess board of dimension N, construct a sequence of moves of a knight such that the knight visits every square once.



en.wiktionary.org

```
// solve problem for fixed N
fun tour(): Tour = choose t:Tour with tour(t);
```

Modeling the Domain

```
// the size of the board (tours exist only for N = 1 or N ≥ 5)
val N:ℕ; axiom rightSize ⇔ N = 1 ∨ N ≥ 5;
type Coord = ℕ[N-1];                    // a board coordinate
type Pos = Record[x:Coord,y:Coord]; // a board position

// the 8 possible jumps of a knight and the corresponding offsets
// (0: right+up, 1: right+down, then counter-clockwise)
type Jump = ℕ[7];
type Offset = ℤ[-2,2];
val x = Map[Jump,Offset](0)
  with [0] =  2 with [1] =  2 with [2] = -1 with [3] =  1
  with [4] = -2 with [5] = -2 with [6] =  1 with [7] = -1;
val y = Map[Jump,Offset](0)
  with [0] =  1 with [1] = -1 with [2] =  2 with [3] =  2
  with [4] = -1 with [5] =  1 with [6] = -2 with [7] = -2;

type Step = ℕ[N·N];
type Tour = Array[N·N-1,Jump];
type Positions = Array[N·N,Pos];
```

Modeling the Core Problem

```
// p are the positions resulting from a tour t
pred positions(t:Tour, p:Positions) ⇔
  p[0] = ⟨x:0,y:0⟩ ∧
  ∀i:Step with i < N·N-1.
    let x1 = p[i].x+x[t[i]], y1 = p[i].y+y[t[i]] in
    p[i+1].x = x1 ∧ p[i+1].y = y1;

// t is a knight's tour starting at (0,0)
pred tour0(t:Tour) ⇔
  ∃pos:Positions with positions(t, pos).
    ∀p:Pos with p.x < N ∧ p.y < N.
      ∃i:Step with i < N·N. pos[i] = p;
```

Modeling the Core Problem (Alternative)

```
// starting at position (x0,y0), t[i].. is a valid tour
pred valid(t:Tour, i:Step, x0:Coord, y0:Coord) ⇔
  if i = N·N-1 then ⊤
  else let x1 = x0+x[t[i]], y1 = y0+y[t[i]] in
       0 ≤ x1 ∧ x1 < N ∧ 0 ≤ y1 ∧ y1 < N ∧ valid(t, i+1, x1, y1);
pred valid(t:Tour) ⇔ valid(t, 0, 0, 0);

// compute the positions resulting from a tour t
fun position(j:Jump, p:Pos):Pos = ⟨x:p.x+x[j], y:p.y+y[j]⟩;
fun positions(t:Tour, i:Step, p:Positions): Positions =
  if i = N·N-1 then p
  else positions(t, i+1, p with [i+1] = position(t[i],p[i]));
fun positions(t:Tour): Positions = positions(t, 0, Array[N*N,Pos](⟨x:0,y:0⟩));

// t is a knight's tour starting at (0,0)
pred tour(t:Tour) ⇔
  valid(t) ∧
  let pos = positions(t) in
  ∀p:Pos with p.x < N ∧ p.y < N. ∃i:Step with i < N·N. pos[i] = p;
```

Computing the Solution Implicitly

```
// determine a knight's tour
fun tour(): Tour = choose t:Tour with tour(t);

Using N=5.
...
Executing tour().
... (waiting)

// also a violation of this theorem determines a tour for size N
// (allows to apply multiple threads in parallel)
theorem notour(t:Tour) ⇔ ¬tour(t);

Executing notour(Array[ℤ]) with all (at least 2^63) inputs.
PARALLEL execution with 4 threads (output disabled).
932235 inputs (593531 checked, 0 inadmissible, 0 ignored, 338704 open)...
1672615 inputs (1283905 checked, 0 inadmissible, 0 ignored, 388710 open)...
2420128 inputs (1925074 checked, 0 inadmissible, 0 ignored, 495054 open)...
...
```

Modeling the Solution as a State Transition System

```
proc tourSystem(): Tuple[Bool,Tour]
  ensures result.1 ⇒ tour(result.2);
{
  var okay:Bool = ⊤;
  var tour:Tour = Array[N·N-1,Jump](0);
  var ps:Positions = Array[N·N,Pos](⟨x:0,y:0⟩);
  var i:Step = 0;
  while okay ∧ i < N·N-1 do
  {
    choose j:Jump with admissible(j, i, ps) then
    {
      tour[i] := j;
      ps[i+1] := position(j,ps[i]);
      i := i+1;
    }
    else okay := ⊥;
  }
  if okay then { print tour ; print positions(tour); }
  return ⟨okay,tour⟩;
}
```

Computing the Solution by a State Transition System

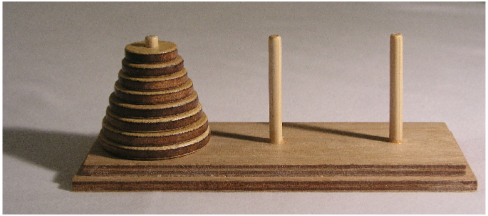
```
// is jump j legal after having jumped to positions ps[0]..ps[i]
pred admissible(j:Jump, i:Step, ps:Positions) ⇔
  let x0 = ps[i].x, y0 = ps[i].y, x1 = x0+x[j], y1 = y0+y[j] in
  0 ≤ x1 ∧ x1 < N ∧ 0 ≤ y1 ∧ y1 < N ∧ ∀k:Step with k < i. ps[k] ≠ ⟨x:x1,y:y1⟩;

// checking the theorem determines one/all tours
theorem notour() ⇔ let r = tourSystem() in ¬r.1; // ⊤ prints all tours
```

```
Using N=5.
...
Executing notour().
[0,0,2,4,7,0,6,5,2,0,1,7,4,2,3,1,6,4,2,2,1,0,7,6]
[[0,0],[2,1],[4,2],[3,4],[1,3],[0,1],[2,2],[3,0],[1,1],[0,3],[2,4],[4,3],
  [3,1],[1,0],[0,2],[1,4],[3,3],[4,1],[2,0],[1,2],[0,4],[2,3],[4,4],[3,2],[4,0]]
ERROR in execution of notour(): evaluation of
  notour
at line 111 in file knight.txt:
  theorem is not true
ERROR encountered in execution.
```

1. Greatest Common Divisor
2. Propositional Satisfiability
3. N Queens
4. Crab and Racoon
5. Goat, Wolf, and Cabbage
6. Knight’s Tour
7. **Tower of Hanoi**
8. Hotel Room Locking
9. A Robotic Controler
10. Steam Boiler Control

Tower of Hanoi

Problem: Given three pegs and N disks of different size that are stacked on the first peg in ascending size, move the disks one by one to the third peg such that never a larger disk is placed on top of a smaller one.



en.wikipedia.org

```
// solve problem for fixed N
fun gameChoose(): Game = choose r:MoveNumber, m:Moves with game(r, m);
```

Modeling the Domain

```
val N:ℕ; axiom notNull ⇔ N > 0; // number of disks
val M = 2^N-1;                   // maximum number of moves required (to be proved ;-)

type Disc = ℕ[N];            // a number 1..N, 0 = None
type Peg  = Array[N, Disc]; // an array of N discs
type PegIndex = ℕ[N-1];      // a peg index denoting a disc
type Board = Array[3, Peg]; // an array of 3 pegs
type BoardIndex = ℕ[2];      // a board index denoting a peg

type Move = Tuple[BoardIndex,BoardIndex]; // a move (from,to)
type Moves = Array[M,Move];               // a sequence of moves
type MoveNumber = ℕ[M];                    // a move number
type Boards = Array[M+1,Board];           // a sequence of boards

type Game = Tuple[MoveNumber,Moves]; // the number of moves and the moves
```

Modeling the Problem

```
pred boards(n:MoveNumber, m:Moves, b:Boards) ⇔ // b are the boards resulting from n moves
  b[0] = iboard ∧
  ∀k:MoveNumber with k < n.
    (∀k0:MoveNumber with k0 ≤ k. legal(b[k0], m[k0])) ⇒ b[k+1] = move(b[k], m[k]);

pred legal(r:MoveNumber, m:Moves) ⇔ // are the first r moves in m legal?
  // choose bs:Boards with boards(r, m, bs) in
  let bs = boards(r, m) in ∀k:MoveNumber with k < r. legal(bs[k], m[k]);

pred end(b:Board) ⇔ // does board b describe the desired end situation?
  ∀k:Disc with 1 ≤ k ∧ k ≤ N. b[2][N-k] = k;

pred game(r:MoveNumber, m:Moves) ⇔ // describe the first r moves of m a complete game?
  let bs = boards(r, m) in
  (∀k:MoveNumber with k < r. legal(bs[k], m[k])) ∧ end(bs[r]);

fun gameChoose(): Game = // the game itself
  // choose r:MoveNumber, m:Moves with game(r, m);
  let r=2^N-1 in ⟨r, choose m:Moves with game(r, m)⟩;
```

Auxiliary Operations

```
// the initial and the zero peg and the initial board
val ipeg:Peg = choose peg: Peg with ∀i:PegIndex. peg[i] = N-i;
val zpeg:Peg = Array[N, Disc](0);
val iboard:Board = Array[3, Peg](zpeg) with [0] = ipeg;

fun pheight(peg:Peg):Disc = // the number of discs on a peg
  if peg[0] = 0 then 0 else max i:PegIndex with peg[i] ≠ 0. 1+i;

fun move(b:Board, m:Move): Board // compute next board from current board b and move m
  requires legal(b, m);
= let h1 = pheight(b[m.1]), h2 = pheight(b[m.2]) in
  b with [m.1] = (b[m.1] with [h1-1] = 0) with [m.2] = (b[m.2] with [h2] = b[m.1][h1-1]);

fun boards(k:MoveNumber, n:MoveNumber, m:Moves, b:Boards): Boards
  requires k ≤ n; decreases n-k;
= if k = n ∨ ¬legal(b[k], m[k]) then b else boards(k+1, n, m, b with [k+1] = move(b[k], m[k]));
fun boards(n:MoveNumber, m:Moves): Boards // the boards resulting from n moves
  ensures boards(n,m,result);
= boards(0, n, m, Array[M+1,Board](iboard));
```

Computing the Solution Implicitly

```
fun gameChoose(): Game = // the game itself
  // choose r:MoveNumber, m:Moves with game(r, m);
  let r=2^N-1 in ⟨r, choose m:Moves with game(r, m)⟩;
```

```
Using N=3.
Computing the truth value of notNull...
Computing the value of ipeg...
Computing the value of zpeg...
Computing the value of iboard...
Computing the value of M...
Type checking and translation completed.
Executing gameChoose().
ERROR in execution of gameChoose(): Exception encountered:
java.lang.StackOverflowError
...
```

Computing the Solution Implicitly (Alternative)

```
// also a violation of this theorem determines a game for N discs
// (allows the application of multiple threads to each candidate)
theorem noGame(m:Moves) ⇔ ¬game(N, 2^N-1, m);

Using N=3.
...
Executing noGame(Array[Tuple[ℤ,ℤ]]) with all 4782969 inputs.
PARALLEL execution with 4 threads (output disabled).
92659 inputs (79173 checked, 0 inadmissible, 0 ignored, 13486 open)...
...
3495580 inputs (3480269 checked, 0 inadmissible, 0 ignored, 15311 open)...
3574537 inputs (3557579 checked, 0 inadmissible, 0 ignored, 16958 open)...
ERROR in execution of noGame([[0,2],[0,1],[2,1],[0,2],[1,0],[1,2],[0,2]]): evaluation of
  noGame
at line 107 in file hanoi.txt:
  theorem is not true
ERROR encountered in execution.
```

Computing the Solution by a Transition System

```
proc game(): Tuple[Bool,MoveNumber,Moves]
  ensures result.1 ⇒ game(result.2,result.3);
{
  var moves:Moves = Array[M,Move](⟨0,0⟩);
  var board:Board = iboard;
  var found:Bool = end(board);
  var i:MoveNumber = 0;
  while ¬found ∧ i < M do
  {
    choose move:Move with legal(board,move);
    moves[i] := move;
    board := move(board, move);
    found := end(board);
    i := i+1;
  }
  if found then print i,moves;
  return ⟨found,i,moves⟩;
}
```

Computing the Solution by a Transition System

```
theorem noGame() ⇔ let r = game() in ¬r.1; // ⊤ to print all games
```

```
Using N=4.
...
Executing noGame().
35609 branches of nondeterministic function noGame().
...
1678560 branches of nondeterministic function noGame().
1715267 branches of nondeterministic function noGame().
15,[[0,1],[0,2],[1,2],[0,1],[2,0],[2,1],[0,1],[0,2],[1,2],[1,0],[2,0],
[1,2],[0,1],[0,2],[1,2]]
ERROR in execution of noGame(): evaluation of
  noGame
at line 132 in file hanoi.txt:
  theorem is not true
ERROR encountered in execution.
```

Computing the Solution by an Algorithm

```
// extend game g by moving n discs from peg i to peg j
proc hanoi(n:Disc, i:BoardIndex, j:BoardIndex, g:Game): Game
  decreases n;
{
  var g0:Game = g;
  if n = 1 then
    g0 := ⟨g0.1+1, g0.2 with [g0.1] = ⟨i,j⟩ ⟩;
  else if n > 1 then
  {
    val k = 3-i-j;
    g0 := hanoi(n-1, i, k, g0);
    g0 := ⟨g0.1+1, g0.2 with [g0.1] = ⟨i,j⟩ ⟩;
    g0 := hanoi(n-1, k, j, g0);
  }
  return g0;
}
```

Computing the Solution by an Algorithm

```
// compute a game for N discs
fun gameCompute(): Game
  ensures game(result.1, result.2);
= let g = ⟨0, Array[M,Move](⟨0,0⟩)⟩ in hanoi(N, 0, 2, g);
```

```
Using N=8.
Executing gameCompute().
Run of deterministic function gameCompute():
Result (4869 ms): [255,[[0,1],[0,2],[1,2],[0,1],[2,0],[2,1],[0,1],[0,2],
[1,2],[1,0],[2,0],[1,2],[0,1],[0,2],[1,2],[0,1],[2,0],[2,1],[0,1],[2,0],
[1,2],[1,0],[2,0],[2,1],[0,1],[0,2],[1,2],[0,1],[2,0],[2,1],[0,1],[0,2],
...
[1,2],[1,0],[2,0],[2,1],[0,1],[0,2],[1,2],[0,1],[2,0],[2,1],[0,1],[0,2],
[1,2],[1,0],[2,0],[1,2],[0,1],[0,2],[1,2],[1,0],[2,0],[2,1],[0,1],[2,0],
[1,2],[1,0],[2,0],[1,2],[0,1],[0,2],[1,2],[0,1],[2,0],[2,1],[0,1],[0,2],
[1,2],[1,0],[2,0],[1,2],[0,1],[0,2],[1,2]]]
Execution completed (4875 ms).
```

1. Greatest Common Divisor
2. Propositional Satisfiability
3. N Queens
4. Crab and Racoon
5. Goat, Wolf, and Cabbage
6. Knight’s Tour
7. Tower of Hanoi
8. **Hotel Room Locking**
9. A Robotic Controler
10. Steam Boiler Control

Hotel Room Locking

Most hotels now issue disposable room keys; when you check out, you can take your key with you. How, then, can the hotel prevent you from reentering your room after it has been assigned to someone else? The trick is *recodable locks*, which have been in use in hotels since the 1980's, initially in mechanical form, but now almost always electronic. The idea is that the hotel issues a new key to the next occupant, which recodes the lock, so that previous keys will no longer work. The lock is a simple, stand-alone unit (usually battery-powered), with a memory holding the current key combination. A hardware device, such as a feedback shift register, generates a sequence of pseudorandom numbers. The lock is opened either by the current key combination, or by its successor; if a key with the successor is inserted, the successor is made to be the current combination, so that the old combination will no longer be accepted. This scheme requires no communication between the front desk and the door locks. By synchronizing the front desk and the door locks initially, and by using the same pseudorandom generator, the front desk can keep its records of the current combinations in step with the doors themselves.

Daniel Jackson: *Software Abstractions*, revised edition, 2012.

Modeling the Protocol

- The protocol is only *probabilistically* correct:
  - In the (unlikely) case that a new key is generated that coincides with the key stored in some lock, the newly issued card may open two different rooms.
  - If we want to verify absolute correctness, we cannot use (pseudo)random keys.
- We may only generate keys that do not coincide with the keys in any locks.
  - But when a newly issued card is inserted into the lock, the locks in the other rooms may have different states than when the card was issued.
  - Thus we cannot compute the same key twice from two different lock states.
- The card holds *two* keys, a new one and the previously generated one.
  - On checkin, the front desk issues a card with a newly generated key and with the previous key; the front desk records the new key as the previous one.
  - If the first key on the card does not match the key in the lock but the second one does, the lock is recoded with the first key.

Some design choices have to be made to model the protocol adequately.

Modeling the Domain

```
// number of rooms, cards, keys
val R: ℕ; val C: ℕ; val K: ℕ;
axiom notZero ⇔ R > 0 ∧ C ≥ R ∧ K > R;

type Room = ℕ[R-1];
type Card = ℕ[C-1];
type Key = ℕ[K-1];
type KeyPair = Tuple[Key,Key];

type Hotel = Record[
  locks:  Array[R,Key],     // key stored in every lock
  cards:  Array[C,KeyPair], // two keys stored in every card
  previous: Array[R,Key],   // previous key assigned to every room
  roomused: Array[R,Bool],  // is room occupied?
  cardused: Array[C,Bool],  // has card been issued?
  assigned: Array[C,Room]   // which room is assigned to every card?
];
```

Auxiliary Operations

```
// key value k is currently in use
pred keyused(k:Key, h:Hotel) ⇔
  (∃r:Room. h.locks[r] = k ∨ h.previous[r] = k) ∨
  (∃c:Card. h.cards[c].1 = k ∨ h.cards[c].2 = k);

// card c opens room r
pred opens(c:Card, r:Room, h:Hotel) ⇔
  h.locks[r] = h.cards[c].1 ∨ h.locks[r] = h.cards[c].2;
```

A Relational Model

```
pred init(h:Hotel) ⇔
  (∀r1:Room, r2:Room with r1 < r2. (h.locks[r1] ≠ h.locks[r2])) ∧
  (∀r:Room. ¬h.roomused[r] ∧ h.locks[r] = h.previous[r]) ∧
  (∀c:Card. ¬h.cardused[c]);

pred checkin(h:Hotel, h0:Hotel) ⇔
  ∃c:Card, k:Key, r:Room with
     ¬h.cardused[c] ∧ ¬keyused(k, h) ∧ ¬h.roomused[r].
   h0 = checkin(c, k, r, h);

pred enter(h:Hotel, h0:Hotel) ⇔
  ∃c:Card, r:Room with h.cardused[c] ∧ opens(c, r, h). h0 = enter(c, r, h);

pred checkout(h:Hotel, h0:Hotel) ⇔
  ∃c:Card with h.cardused[c]. h0 = checkout(c, h);

pred next(h:Hotel, h0:Hotel) ⇔ checkin(h, h0) ∨ enter(h, h0) ∨ checkout(h, h0);
```

An Action-Oriented Model

```
rectype (1) Action = checkin(Card,Key,Room) | enter(Card,Room) | checkout(Card);

pred admissible(a:Action, h:Hotel) ⇔
  match a with
  {
    checkin(c:Card, k:Key, r:Room) -> ¬h.cardused[c] ∧ ¬keyused(k, h) ∧ ¬h.roomused[r];
    enter(c:Card, r:Room) -> h.cardused[c] ∧ opens(c, r, h);
    checkout(c:Card) -> h.cardused[c];
  };
fun next(a:Action, h:Hotel): Hotel =
  match a with
  {
    checkin(c:Card, k:Key, r:Room) -> checkin(c, k, r, h);
    enter(c:Card, r:Room) -> enter(c, r, h);
    checkout(c:Card) -> checkout(c, h);
  };

theorem equiv(h:Hotel,h0:Hotel) ⇔ next(h,h0) ⇔ ∃a:Action with admissible(a,h). h0 = next(a,h);
```

An Action-Oriented Model

```
val N:ℕ;
type Index = ℕ[N];

proc run(h0:Hotel, n:Index): Hotel
  requires init(h0);
{
  var h:Hotel = h0;
  var as: Array[N,Action] = Array[N,Action](Action!enter(0,0));
  var i:Index = 0;
  if ¬safe(h) then { print as; print "{1}: {2}", i, h; assert ⊥; }
  while i < n do
  {
    choose a:Action with admissible(a, h);
    h := next(a, h);
    as[i] := a;
    i := i+1;
    if ¬safe(h) then { print as; print "{1}: {2}", i, h; assert ⊥; }
  }
  return h;
}
```

The State Transitions

```
// new guest is given card c with fresh key k and asssigned room r
fun checkin(c:Card, k:Key, r:Room, h:Hotel): Hotel
= h with .cards = h.cards with [c] = ⟨k,h.previous[r]⟩
    with .previous = h.previous with [r] = k
    with .roomused = h.roomused with [r] = ⊤
    with .cardused = h.cardused with [c] = ⊤
    with .assigned = h.assigned with [c] = r;

// guest with card c enters room r
fun enter(c:Card, r:Room, h:Hotel): Hotel
= if h.locks[r] = h.cards[c].1
    then h
    else h with .locks  = h.locks with [r] = h.cards[c].1;

// guest with card c checks out, we may reuse its card and room
fun checkout(c:Card, h:Hotel): Hotel
= h with .cardused = h.cardused with [c] = ⊥
    with .roomused = h.roomused with [h.assigned[c]] = ⊥;
```

The Safety Property

How, then, can the hotel prevent you from reentering your room after it has been assigned to someone else?

- Reformulation:
  - If a room is assigned to a guest, no other guest can enter the room.
  - If a card can open a room, no other card can open the room.

```
pred safe(h:Hotel) ⇔
  ¬∃r:Room, c1: Card, c2: Card. opens(c1, r, h) ∧ opens(c2, r, h) ∧ c1 ≠ c2;
```

```
Using R=2. Using C=2. Using K=5. Using N=1.
Executing run0().
[0:0:0:[1,2,0]]
1: [[0,1],[[0,0],[2,0]],[2,1],[true,false],[false,true],[0,0]]
ERROR in execution of run0(): assertion failed
```

Another *not issued* card may actually open the room.

The Safety Property

If an issued card can open a room, no other issued card can open the room.

```
pred safe(h:Hotel) ⇔
  ¬∃r:Room, c1: Card, c2: Card.
    h.cardused[c1] ∧ opens(c1, r, h) ∧ h.cardused[c2] ∧ opens(c2, r, h) ∧ c1 ≠ c2
```

```
Using R=2.
Using C=2.
Using K=5.
Computing the truth value of notZero...
Using N=10.
Type checking and translation completed.
Executing run0().
40817 branches of nondeterministic function run0().
...
549628 branches of nondeterministic function run0().
Execution completed (28482 ms).
```

With two rooms/cards and five key values, the protocol is safe for 10 steps.

Additional Properties

Every issued card can open some room.

```
pred safe(h:Hotel) ⇔ ... ∧ (∀c:Card. h.cardused[c] ⇒ ∃r:Room. opens(c, r, h))
```

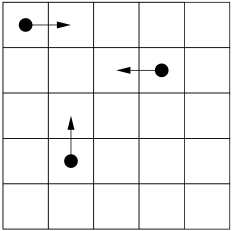
```
Using R=2.
Using C=2.
Using K=5.
Computing the truth value of notZero...
Using N=3.
Type checking and translation completed.
Executing run0().
[0:0:0:[0,2,0],0:2:0:[0],0:0:0:[0,3,0]]
3: [[0,1],[[3,2],[1,1]],[3,1],[true,false],[true,false],[0,0]]
ERROR in execution of run0(): assert ⊥;
```

If a guest checks out without having entered the room, the lock is not recoded and the next issued card cannot open that room; thus the room becomes inaccessible. To open the room again, an additional “master key” has to be introduced.

1. Greatest Common Divisor
2. Propositional Satisfiability
3. N Queens
4. Crab and Racoon
5. Goat, Wolf, and Cabbage
6. Knight’s Tour
7. Tower of Hanoi
8. Hotel Room Locking
9. **A Robotic Controler**
10. Steam Boiler Control

Example: A Robotic Controler

An grid in which multiple robots move around.



- System: each robot moves one cell in a selected direction.
- Safety: the robots shall not collide with the walls or with each other.

Our task is to model an adequate control software for each robot: given the current situation of the system, compute a safe direction for the movement of the robot.

Modeling the System Execution

The problem becomes non-trivial if we also consider the *inertia* of robots.

```
proc system(x0: Positions, y0: Positions, d0: Directions): ()
  requires init(x0, y0, d0);
{
  var x: Positions = x0; var y: Positions = y0; var d: Directions = d0;
  for var i:ℕ[N] := 0; i < N; i := i+1 do
  {
    choose r: Robot;
    x := x with [r] = moveX(x[r], d[r]);
    y := y with [r] = moveY(y[r], d[r]);
    assert noCollision(x, y);
    choose dr: Direction with nextDir(x, y, d, r, dr);
    d[r] := dr;
  }
}
```

Each robot moves in the direction already chosen in the *previous* step.

Modeling the Domain

```
val R:ℕ; // number of robots
val P:ℕ; // number of positions

axiom notzero ⇔ R ≥ 1 ∧ P ≥ 1;

type Robot = ℕ[R-1];
type Position = ℕ[P-1];
enumtype Direction = Stop | Left | Right | Up | Down;

type Positions = Array[R,Position];
type Directions = Array[R,Direction];
```

Constraining the Initial State

```
// the desired safety property of the system:
// no two robots are at the same position
pred noCollision(x:Positions, y:Positions) ⇔
  ∀r1:Robot, r2:Robot with r1 < r2. x[r1] ≠ x[r2] ∨ y[r1] ≠ y[r2];

// the initial state condition of the system:
// robots are at different positions and do not yet move
pred init(x:Positions, y:Positions, d:Directions) ⇔
  noCollision(x, y) ∧ ∀r:Robot. d[r] = Direction!Stop;
```

Moving to the Next Position

```
// may robot at position x,y move into direction d?
pred mayMove(x:Position, y:Position, d:Direction) ⇔
  match d with
  {
    Left -> x > 0; Right -> x < P-1; Up -> y > 0; Down -> y < P-1; Stop -> ⊤;
  };

// the new positions if robot moves into direction d
fun moveX(x:Position, d:Direction):Position =
  match d with
  {
    Left -> x-1; Right -> x+1; Stop -> x; Up -> x; Down  -> x;
  };
fun moveY(y:Position, d:Direction):Position =
  match d with
  {
    Up -> y-1; Down -> y+1; Stop -> y; Left -> y; Right  -> y;
  };
```

Choosing the Next Direction

```
// robot r is at position xr,yr or can move there by its chosen direction
pred moveTo(x:Positions, y:Positions, d:Directions,
  r:Robot, xr:Position, yr:Position) ⇔
  (xr = x[r] ∧ yr = y[r]) ∨
  (mayMove(x[r],y[r],d[r]) ∧ xr = moveX(x[r],d[r]) ∧ yr = moveY(y[r],d[r]))
;

// any robot different from r can move to position xr, yr
pred anyTo(x:Positions, y:Positions, d:Directions,
  r:Robot, xr:Position, yr:Position) ⇔
  ∃r0: Robot with r0 ≠ r. moveTo(x, y, d, r0, xr, yr)
;

// the relation between the current system state and the new direction dr of robot r
pred nextDir(x:Positions, y:Positions, d: Directions, r:Robot, dr:Direction) ⇔
  mayMove(x[r],y[r],dr) ∧
  let xr = moveX(x[r],dr), yr = moveY(y[r],dr) in
  ¬anyTo(x,y,d,r,xr,yr)
;
```

Verifying the Safety of the System

Checking the safety for three steps.

```
Using R=3.
Using P=5.
Computing the truth value of notzero...
Using N=3.
Type checking and translation completed.
Executing system(Array[ℤ],Array[ℤ],Array[Direction[...]]) with all 1953125 inputs.
PARALLEL execution with 4 threads (output disabled).
486 inputs (310 checked, 130 inadmissible, 0 ignored, 46 open)...
881 inputs (624 checked, 207 inadmissible, 0 ignored, 50 open)...
1062 inputs (834 checked, 215 inadmissible, 0 ignored, 13 open)...
1403 inputs (1070 checked, 225 inadmissible, 0 ignored, 108 open)...
1697 inputs (1321 checked, 256 inadmissible, 0 ignored, 120 open)...
...
```

Extremely slow due to combinatorial explosion of choices.

Verification by Inductive System Invariant

```
// the system invariant
pred inv(x:Positions, y:Positions, d:Directions) ⇔
  noCollision(x, y) ∧
  ∀r:Robot. mayMove(x[r], y[r], d[r]) ∧
    let xr = moveX(x[r], d[r]), yr = moveY(y[r], d[r]) in
    ¬anyTo(x, y, d, r, xr, yr);

// the system invariant implies the desired safety property
theorem invIsStrongEnough(x:Positions, y:Positions, d:Directions) ⇔
  inv(x, y, d) ⇒ noCollision(x, y);

// the invariant holds in the initial state and is preserved by the transition relation
theorem invHoldsInitially(x:Positions, y:Positions, d:Directions) ⇔
  init(x, y, d) ⇒ inv(x, y, d);
theorem invIsPreserved(x:Positions, y:Positions, d:Directions) ⇔
  inv(x, y, d) ⇒
    ∀x0:Positions, y0:Positions, d0:Directions.
    next(x, y, d, x0, y0, d0) ⇒ inv(x0, y0, d0);
```

Verification of safety for an infinite number of steps.

Checking the Induction Step

```
// the relationship between the prestate of the system and its poststate
pred next(x:Positions, y:Positions, d:Directions,
  x0:Positions, y0:Positions, d0:Directions) ⇔
  ∃r:Robot.
    x0 = moveX(x, r, d[r]) ∧ y0 = moveY(y, r, d[r]) ∧
    ∃dr: Direction with nextDir(x0, y0, d, r, dr).
      d0 = d with [r] = dr;
```

```
Executing system(Array[ℤ],Array[ℤ],Array[Direction[...]]) with all 1953125 inputs.
PARALLEL execution with 4 threads (output disabled).
327 inputs (0 checked, 8 inadmissible, 0 ignored, 319 open)...
327 inputs (4 checked, 10 inadmissible, 0 ignored, 313 open)...
327 inputs (4 checked, 10 inadmissible, 0 ignored, 313 open)...
327 inputs (8 checked, 12 inadmissible, 0 ignored, 307 open)...
...
```

Still very slow due to large state space for each choice.

Checking the Induction Step (Alternative)

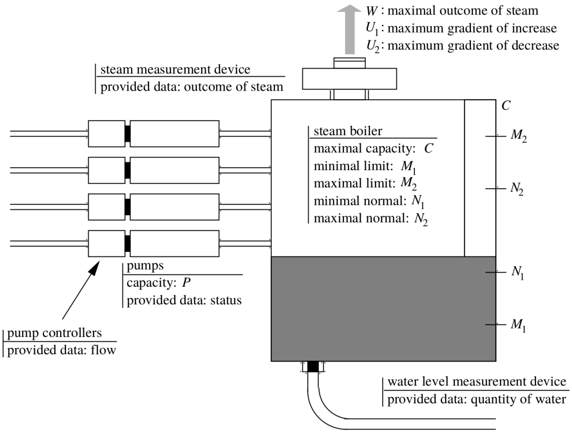
```
theorem invIsPreserved(x:Positions, y:Positions, d:Directions) ⇔
  inv(x, y, d) ⇒
  // ∀x0:Positions, y0:Positions, d0:Directions.
  // next(x, y, d, x0, y0, d0) ⇒ inv(x0, y0, d0);
  ∀r:Robot.
    let x0 = x with [r] = moveX(x[r], d[r]), y0 = y with [r] = moveY(y[r], d[r]) in
    ∀dr:Direction with nextDir(x0, y0, d, r, dr).
      let d0 = d with [r] = dr in
      inv(x0, y0, d0);
```

```
Executing invIsPreserved(Array[ℤ],Array[ℤ],Array[Direction[...]]) with all 1953125 inputs.
PARALLEL execution with 4 threads (output disabled).
19137 inputs (14588 checked, 0 inadmissible, 0 ignored, 4549 open)...
...
1925886 inputs (1917852 checked, 0 inadmissible, 0 ignored, 8034 open)...
Execution completed for ALL inputs (136282 ms, 1953125 checked, 0 inadmissible).
```

System is safe for infinitely many steps.

1. Greatest Common Divisor
2. Propositional Satisfiability
3. N Queens
4. Crab and Racoon
5. Goat, Wolf, and Cabbage
6. Knight’s Tour
7. Tower of Hanoi
8. Hotel Room Locking
9. A Robotic Controler
10. **Steam Boiler Control**

The Steam Boiler Control Specification Problem



Mickael Kerboeuf et al: *Specification and Verification of a Steam-Boiler with Signal-Coq*, 2010

Control the boiler such that the water remains at a safe level.

System Components

- Boiler with capacity C.
  - □ Water level must be always between M_1 and M_2 with $0 \leq M_1 \leq M_2 \leq C$.
  - □ Water level shall be normally between N_1 and N_2 with $M_1 \leq N_1 \leq N_2 \leq M_2$.
  - □ Sensor reports actual quantity q of water in boiler.
- Boiler exit with maximum quantity W of steam.
  - □ Sensor reports actual quantity w of steam currently produced.
  - □ Quantity is at most incremented by U_1 and decremented by U_2 in each cycle.
- 4 pumps with capacity P each.
  - □ 4 controllers to switch each pump on or off.
  - □ Switching off takes place immediately, switching on takes effect in *next* cycle.
- Boiler valve is open or closed.
  - □ Used only initially to get rid of too much water.

Software must control status of valve and of pumps in each cycle.

System Operation

System is in one of the modes “init”, “run”, or “stop”; it operates in a sequence of “sensor/control cycles”.

- Initial mode “init”: if sensor reports some steam production, it is defective, thus immediately switch to mode “stop”; if there is too much water, open valve; if there is too little water, open pump; otherwise switch to mode “run”.
- Normal model “run”: try to maintain water level between N_1 and N_2 by switching pumps on or off; if the water level falls below M_1 or exceeds M_2, then switch to mode “stop”.
- Emergency mode “stop”: system has entered a critical state and is stopped.

A very simple subset (mostly without component failures) of the “steam boiler control specification problem” (Jean-Raymond Abrial, et al., 1996), a classical case study in formal modeling.

Model Parameters

```
val C = 40;  // maximal capacity of boiler
val M1 = 3;  // minimal critical limit
val M2 = 32; // maximal critical limit
val N1 = 15; // minimal normal limit
val N2 = 20; // maximal normal limit
val W = 4;   // maximal steam output
val U1 = 1;  // maximum gradient of its increase
val U2 = 2;  // maximum gradient of its decrease
val P = 2;   // nominal capacity of each pump
val N = 4;   // number of pumps
```

Sample values (that will turn out to be safe for at least 7 cycles).

System Operation

```
val S:ℕ;
proc boiler(): ()
{
  var i:ℕ[S] = 0;
  var b:Boiler = init();
  while i < S ∧ ¬stopped(b) do
  {
    val c = analyze(b);
    // b = choose b0:Boiler with control(b, c, b0);
    b := control(b, c);
    // if ¬safe(b) then print ⟨i,b⟩;
    assert safe(b);
    i := i+1;
  }
  // print ⟨i,b⟩;
}
```

Verification of S (non-deterministic) execution steps.

Core Types

```
type Boiler = Record[
 s: System,   // the system mode
 v: Valve,    // the status of the valve
 m: Modes,    // the actual current pump modes
 q: Water,    // the current quantity of water in the boiler
 w: Steam     // the quantity of steam currently produced
];

type Control = Record[
  s: System,  // the new system mode
  v: Valve,   // the new status of the valve
  r: Requests // the requested pump states
];
```

The state of the boiler and the control decision of the software.

Basic Types

```
type Pump = ℕ[N-1];  // pump index
type System = ℕ[2];  // system mode (0:init, 1:run, 2:stop)
type Valve = Bool;   // valve state (false:closed, true:open)
type Mode = ℕ[2];    // pump mode (0:off, 1:on, 2:switch)
type Modes = Map[Pump,Mode]; // mode of pumps
type Water = ℕ[C];   // water in boiler
type Steam = ℕ[W];   // quantity of steam
type Request = Bool; // true if pump mode is to be switched
type Requests = Map[Pump,Request]; // requests for mode switches
type Flow = ℕ[P];    // a pump flow
```

The components of the boiler state and of the control decision.

Initial and Final State

```
// an arbitrary initial boiler state (may be even critical)
fun init(): Boiler =
  choose b:Boiler with
  b.s = 0 ∧ b.v = ⊥ ∧ b.m = Map[Pump,Mode](0) ∧ b.w = 0;

// an initial boiler state with water quantity q
fun init1(q:Water): Boiler =
  choose b:Boiler with
  b.s = 0 ∧ b.v = ⊥ ∧ b.m = Map[Pump,Mode](0) ∧ b.w = 0 ∧
  b.q = q;

// system is stopped
pred stopped(b:Boiler) ⇔ b.s = 2;
```

Initial and final state conditions.

Safety Condition

```
// the safety condition of the boiler
pred safe(b:Boiler) ⇔
  // in initial state, no stream is produced
  (b.s = 0 ⇒ b.w = 0) ∧
  // valve may be only open in initial state
  (b.v ⇒ b.s = 0) ∧
  // in running system, water level must not be critical
  (b.s = 1 ⇒ M1 ≤ b.q ∧ b.q ≤ M2);
```

Condition must hold at all system states.

Applying a Control Decision (Relational Version)

```
// b0 is a possible new state of a boiler with previous state b
// determined by control decision c
pred control(b:Boiler, c:Control, b0:Boiler) ⇔
  // quantity of water leaving through valve
  ∃v:Flow with if ¬c.v then v = 0 else v > 0.
  // number of pumps currently open
  let n = #i:Pump with b.m[i] = 1 in
  // quantity of water entering from pumps
  ∃p:ℕ[N*P] with n ≤ p ∧ p ≤ n*P.
  // the new quantity of water from the choices
  let q = let q = b.q-(v+b.w)+p in if q < 0 then 0 else q in
  // new amount of steam produced
  ∃w:Steam with
    if b.s = 0 then w = 0 else 1 ≤ w ∧ b.w-U2 ≤ w ∧ w ≤ b.w+U1.
  // new pump modes
  let m = modes(b.m, c.r) in
  // new boiler state
  (b0.s = c.s ∧ b0.v = c.v ∧ b0.m = m ∧ b0.q = q ∧ b0.w = w);
```

Applying a Control Decision (Functional Version)

```
// determine a possible new state of a boiler with previous state b
// from control decision c
fun control(b:Boiler, c:Control): Boiler =
  // quantity of water leaving through valve
  choose v:Flow with if ¬c.v then v = 0 else v > 0 in
  // number of pumps currently open
  let n = #i:Pump with b.m[i]=1 in
  // quantity of water entering from pumps
  choose p:ℕ[N*P] with n ≤ p ∧ p ≤ n*P in
  // new quantity of water from the choices
  let q = let q0 = b.q-(v+b.w)+p in if q0 < 0 then 0 else q0 in
  // new amount of steam produced
  choose w:Steam with
    if b.s = 0 then w = 0 else 1 ≤ w ∧ b.w-U2 ≤ w ∧ w ≤ b.w+U1 in
  // new pump modes
  let m = modes(b.m, c.r) in
  // new boiler state
  b with .s = c.s with .v = c.v with .m = m with .q = q with .w = w;
```

Making a Control Decision

```
// analyze state b of boiler and determine a control decision
fun analyze(b:Boiler): Control
  requires b.s ≠ 2;           // not run in emergency mode
= if b.q < M1 ∨ b.q > M2 then // quantity of water is critical, emergency stop
    ⟨s:2, v:b.v, r:none⟩
  else if b.s = 0 then        // initial state with non-critical quantity of water
    if b.w ≠ 0 then           // steam sensor has failed, emergency stop
      ⟨s:2, v:b.v, r:none⟩
    else if b.q > N2 then     // too much water, open valve
      ⟨s:b.s, v:⊤, r:none⟩
    else if b.q < N1 then     // too little water, make sure valve is closed and open some pump
      ⟨s:b.s, v:⊥, r:requests(b.m, b.q)⟩
    else                      // safe state has been reached,
      ⟨s:1, v:⊥, r:none⟩      // make sure that valve is closed and go to running mode
  else                        // running state with non-critical quantity of water
    ⟨s:b.s, v:b.v, r:requests(b.m, b.q)⟩;
```

Determining the Pump Requests

```
// request to not switch anything
val none = Map[Pump,Request](⊥);

// compute requests from pump modes m and quantity q of water
fun requests(m:Modes, q:Water): Requests =
  let m0 = modes(m) in  // previously requests to open pumps have now taken effect
  if q < N1 then        // request some additional closed pump, if possible
    choose i:Pump with m0[i] = 0 in
      none with [i] = ⊤
    else
      none
  else if q > N2 then   // close some open pump, if possible
    choose i:Pump with m[i] = 1 in
      none with [i] = ⊤
    else
      none
  else                  // everything is fine
    none;
```

Determining the Pump Modes

```
// determine next modes of pumps
fun modes(m:Modes): Modes =
  choose m0:Modes with
    ∀i:Pump. m0[i] = if m[i] = 2 then 1 else m[i];

// determine next modes of pumps after applying requests
fun modes(m:Modes, r:Requests): Modes =
  let m0 = modes(m) in
  choose m1:Modes with
    ∀i:Pump.
      if ¬r[i] then
        m1[i] = m0[i]
      else if m0[i] = 1 then
        m1[i] = 0  // switch off takes immediate effect
      else
        m1[i] = 2; // switch on takes effect after next step
```

Verifying the System

```
Using S=7.
Type checking and translation completed.
Executing boiler().
12620 branches of nondeterministic function boiler().
25346 branches of nondeterministic function boiler().
38029 branches of nondeterministic function boiler().
50971 branches of nondeterministic function boiler().
63750 branches of nondeterministic function boiler().
76435 branches of nondeterministic function boiler().
89224 branches of nondeterministic function boiler().
Execution completed (15292 ms).
```

The system is safe for all possible executions with 7 cycles.

Verifying the System

```
val N2 = 21; // maximal normal limit

Using S=100.
Type checking and translation completed.
Executing boiler().
[99,[1,false,[0,0,0,0],33,1]]
ERROR in execution of boiler(): evaluation of
  assert safe(b);
at line 196 in file boiler.txt:
  assertion failed
ERROR encountered in execution.
```

For $N_2 = 21$, there is an execution with 99 cycles that leads to the water level exceeding the maximal critical limit (to ensure the safety of the steam boiler, it may be necessary to close more than one pump).