DYNALLOY: AN EXTENSION OF ALLOY FOR WRITING AND ANALYZING BEHAVIOURAL MODELS

Germán Regis | César Cornejo | Simón Gutiérrez Brida | Mariano Politano | Fernando Raverta | Pablo Ponzio | Nazareno Aguirre | Juan Pablo Galeotti | Marcelo Frias

Universidad Nacional de Río Cuarto
Universidad de Buenos Aires
Instituto Tecnológico Buenos Aires

Workshop on the Future of Alloy
EXAMPLE - RIVER CROSSING PUZZLE

A farmer wants to cross a river with a fox, a chicken, and a sack of grain. The farmer's boat can only carry him and one other item at a time. How can the farmer successfully cross the river without losing any of his items?
EXAMPLE - RIVER CROSSING PUZZLE

near

Fa
Ch
Fox
Gr

far

near

far

Fa
Ch
Fox
Gr
abstract sig Object { eats: set Object}

one sig Farmer, Fox, Chicken, Grain extends Object {}

fact { eats = Fox->Chicken + Chicken->Grain }

sig State { near, far: set Object }
RIVER CROSS - DYNAMIC BEHAVIOR

State Change

Ordering

1 2 3 4 5 6 7 8
open util/ordering[State]
sig State { near, far: set Object }

fact { first.near = Object && no first.far}
open util/ordering[State]

sig State { near, far: set Object }

fact { first.near = Object && no first.far}

pred crossRiver[from, from', to, to': set Object]
{ one x: from | {
    from' = from - x - Farmer - from'.eats &&
    to' = to + x + Farmer }
}
open util/ordering[State]

sig State { near, far: set Object }

fact { first.near = Object && no first.far}

pred crossRiver[from, from', to, to': set Object]
{ one x: from | {
  from' = from - x - Farmer - from'.eats &&
  to' = to + x + Farmer }
}

fact { all s: State, s': s.next | { Farmer in s.near =>
  crossRiver[s.near, s'.near, s.far, s'.far] else crossRiver[s.far, s'.far, s.near, s'.near] } }
open util/ordering[State]
sig State { near, far: set Object }

fact { first.near = Object && no first.far}

pred crossRiver[from, from', to, to': set Object]
{ one x: from | { from' = from - x - Farmer - from'.eats && to' = to + x + Farmer } }

fact { all s: State, s': s.next | { Farmer in s.near => crossRiver[s.near, s'.near, s.far, s'.far] else crossRiver[s.far, s'.far, s.near, s'.near] } }

run { last.far = Object } for 8 States

SATisfying valuations of the predicate are solutions to the puzzle
open util/ordering[State]

sig State { near, far: set Object }

fact { first.near = Object && no first.far}

pred crossRiver[from, from', to, to': set Object]
{ one x: from | {
from' = from − x − Farmer − from'.eats &&
to' = to + x + Farmer }
}

fact { all s: State, s': s.next | { Farmer in s.near =>
crossRiver[s.near, s'.near, s.far, s'.far]
else crossRiver[s.far, s'.far, s.near, s'.near] }
}

run { last.far = Object } for 8 States

SATisfying valuations of the predicate are solutions to the puzzle
Execution traces are indirectly defined through:

- **Atomic Actions** (State Change)
- **Programs** (imperative style & nondeterminism)

Assumptions | Test ? | Choice + | Sequential Composition ; | Iteration *

DYNALLOY = ALLOY + DYNAMIC LOGIC
open util/ordering[State]
sig State { near, far: set Object }
fact { first.near = Object && no first.far}
pred    crossRiver[from,from’, to, to’: set Object]
{ one x: from | {
    from’ = from – x – Farmer – from’.eats &&
    to’ = to + x + Farmer }
}

fact { all s: State, s’: s.next | { Farmer in s.near =>
    crossRiver[s.near, s’.near, s.far, s’.far]
    else crossRiver[s.far, s’.far, s.near, s’.near] } }

run { last.far = Object } for 8 States
sig State { near, far: set Object }

fact { first.near = Object && no first.far }

pred crossRiver[from,from’, to, to’: set Object]
{ one x: from | {
    from’ = from – x – Farmer – from’.eats &&
    to’ = to + x + Farmer
}
}

fact { all s: State, s’: s.next | { Farmer in s.near =>
    crossRiver[s.near, s’.near, s.far, s’.far]
    else crossRiver[s.far, s’.far, s.near, s’.near] } }

run { last.far = Object } for 8 States
**RIVER CROSS - DYNALLOY SPECIFICATION**

`sig` State `{ near, far: set Object }`

`pred` crossRiver[from, from', to, to' : set Object] {
    one x: from | {
        from' = from - x - Farmer - from’.eats &&
        to' = to + x + Farmer
    }
}

`fact` { all s: State, s': s.next | {
    Farmer in s.near =>
    crossRiver[s.near, s'.near, s.far, s'.far]
    else crossRiver[s.far, s'.far, s.near, s'.near] }
}

`run` { last.far = Object } for 8 States
RIVER CROSS - DYNALLOY SPECIFICATION

sig State { near, far: set Object }

action crossRiver[from, to : set Object]
pre { Farmer in from }
post { one x: from | {
  from' = from - x - Farmer - from'.eats &&
  to' = to + x + Farmer }
}

fact { all s: State, s': s.next | { Farmer in s.near =>
  crossRiver[s.near, s'.near, s.far, s'.far]
  else crossRiver[s.far, s'.far, s.near, s'.near] } }

run { last.far = Object } for 8 States
RIVER CROSS - DYNALLOY SPECIFICATION

sig State { near, far: set Object }

action crossRiver[from, to : set Object]
pre { Farmer in from }
pot{ one x: from | {
    from' = from - x - Farmer - from'.eats &&
    to' = to + x + Farmer }
}

program solvePuzzle[near, far: set Object] {
    assume (Object in near && no far);
    (crossRiver[near, far] + crossRiver[far, near])*;
    [Object in far]?
}

run { last.far = Object } for 8 States
**RIVER CROSS - DYNALLOY SPECIFICATION**

`sig State { near, far: set Object }

action crossRiver[from, to : set Object]`  
`pre { Farmer in from }
`  
`post { one x: from | {
  from' = from - x - Farmer - from’.eats &&
  to' = to + x + Farmer }
`  
`}

`program solvePuzzle[near, far: set Object] {
  assume (Object in near && no far);
  (crossRiver[near, far] + crossRiver[far, near])*;
  [Object in far]?
`  
`}

`run solvePuzzle for 4 lurs 8`
RIVER CROSS - DYNALLOY
PARTIAL CORRECTNESS ASSERTIONS

{ precondition }

PROGRAM

{ postcondition }
assert noResurrection[near, far: set Object, x: Object] {
  pre { no (near & far) }
  prog {
    (crossRiver[near, far] + crossRiver[far, near])*
    [x !in (near+far)] ?
    (crossRiver[near, far] + crossRiver[far, near])*
  }
  post { x !in (near'+far') }
}

check noResurrection for 4 lurs 8
DYNALLOY FEATURES

module examples/case_studies/crossRiver

abstract sig Object {
  eats: set Object
}

one sig Farmer, Fox, Chicken, Grain extends Object {

fact {
  eats = Fox->Chicken + Chicken->Grain
}

act crossRiver[from, to: set Object] {
  pre { Farmer in from }
  post { one x: from | from' = from - x - Farmer - from'.eats && to' =

/*==========*/
assert SolvePuzzleAsCorrectnessAssertion[near: set Object, far: set Obj
  pre { Object in near && no far}
  prog { (crossRiver[near, far] + crossRiver[far, near])* }
  post { Object in far }
}

check SolvePuzzleAsCorrectnessAssertion for 4 expect 1 lurs 8

/*==========*/
prog SolvePuzzleAsProgramRun[near: set Object, far: set Object] var [i:
  assume (Object in near && no far);
  (crossRiver[near, far] + crossRiver[far, near])*;
  [ Object in far ]?
}
DYNALLOY FEATURES

- Completely integrated into Alloy Analyzer
- Fully compatible with standard Alloy, produces detailed compile-time error reports
- Supports abstract syntax of programs, as well as imperative programming constructs (assignment, while loops, subprogram calls, …)
- Trace visualization (in the style of a program debugger)
DYNALLOY FEATURES

• Completely integrated into Alloy Analyzer
• Fully compatible with standard Alloy, produces detailed compile-time error reports
• Supports abstract syntax of programs, as well as imperative programming constructs (assignment, while loops, subprogram calls, …)
• Trace visualization (in the style of a program debugger)

Next release:
• Efficient characterization of traces using skolemization
• Efficient real and integer arithmetical representation
• Control flow graph visualization for analyzing execution traces