

# VALIDATING MATHEMATICAL THEORIES AND ALGORITHMS WITH RISCAL



Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC)

Johannes Kepler University, Linz, Austria



# The RISC Algorithm Language (RISCAL)

A language and software system for investigating finite mathematical models.

- Formulation of mathematical theories and theorems.
- Formulation and specification of (also non-deterministic) algorithms.
- Rooted in strongly typed first order logic and set theory.
- All types are finite (with sizes determined by model parameters).
- All formulas are automatically decidable by model checking.
- Correctness of all algorithms is automatically decidable by model checking.

Checking in some model of fixed size before proving in models of arbitrary size.

# RISCAL Specifications

```
val n: ℕ;
type LiteralBase = ℤ[-n,n];
type Literal = LiteralBase with value ≠ 0;
...
pred satisfiable(f:Formula) ⇔ ∃v:Valuation. satisfies(v,f);
pred valid(f:Formula) ⇔ ∀v:Valuation. satisfies(v,f);
fun not(f: Formula):Formula = { c | c:Clause with ∀d∈f. ∃l∈d. -l∈c };
...
theorem notValid(f:Formula) ⇔ valid(f) ⇔ ¬satisfiable(not(f));
...
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));
```

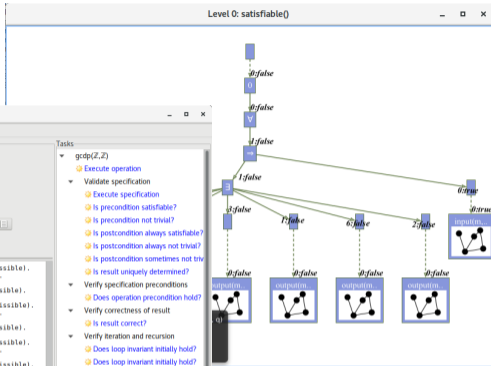
# The RISCAL Software

The screenshot shows the RISCAL software interface. On the left is the source code editor with the following code:

```
1 // .....  
2 // Computing the greatest common divisor by the Euclidean Algorithm  
3 // .....  
4  
5 val N: %  
6 type nat = %[N];  
7  
8 pred divides(m:nat,n:nat) = ∃p:nat. m*p = n;  
9  
10 fun gcd(m:nat,n:nat): nat  
11   requires m ≠ 0 ∧ n ≠ 0;  
12   = choose result:nat with  
13     divides(result,m) ∧ divides(result,n) ∧  
14     -∃r:nat. divides(r,m) ∧ divides(r,n) ∧ r > result;  
15  
16 val gnat = gcd(N,N-1);  
17  
18 theorem gcd0(m:nat) = m=0 → gcd(m,0) = m;  
19 theorem gcd1(m:nat,n:nat) = m ≠ 0 ∧ n = 0 → gcd(m,n) = gcd(n,m);  
20 theorem gcd2(m:nat,n:nat) = 1 ≤ m ∧ n ≤ m → gcd(m,n) = gcd(m/N,n);  
21  
22 proc gcd(m:nat,n:nat): nat  
23   requires m ≠ 0 ∧ n ≠ 0;  
24   ensures result = gcd(m,n);  
25 {  
26   var a:nat = m;  
27   var b:nat = n;  
28   while a ≥ 0 ∧ a ≤ b do  
29     invariant a = 0 ∨ b ≠ 0;  
30     invariant gcd(a,b) = gcd(oldest_a,oldest_b);  
31     decreases a+b;  
32   {  
33     if a ≥ b then  
34       a = a-b;  
35     else  
36       b = b-a;  
37   }  
38   return if a = 0 then b else a;  
39 }  
40  
41 fun gcdf(m:nat,n:nat): nat  
42   requires m ≠ 0 ∧ n ≠ 0;  
43   ensures result = gcd(m,n);  
44   decreases m+n;
```

The middle panel shows analysis options: Translation (Nonterminism, Default Value: 0, Other Values:), Execution (Silent, Inputs, Per Mile, Branches), Visualization (Trace, Tree, Width: 800, Height: 600), Parallelism (Multi-Threaded, Threads: 4, Distributed Servers), and Operation (gcdp(Z,Z)).

The right panel shows execution results for various inputs, including warnings about nondeterministic branches and completion times for different input sizes.



Automatic checking of theorems, algorithms, and verification conditions; visual explanation of formula values.