LOGICAL MODELS OF PROBLEMS AND COMPUTATIONS

Theory and Software

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

JOHANNES KEPLER
UNIVERSITY LINZ

Logical Models of Problems and Computations

What is the purpose of logical modeling?

- Precisely describe the problem to be solved.
  - Clarification of mind, resolution of ambiguities.
  - Specification of program to be developed.
- Software-supported analysis of the problem and its solution.
  - Validation of specification.
  - Validation/verification of solution.
  - Interactive/automatic provers and model checkers.
- Automatic computation of solution respectively simulation of execution.
  - Logical solvers (SMT: Satisfiability Modulo Theories) and theorem provers.
  - Perhaps: rapid prototyping of a later manually written program.

To profit from software, we need computer-understandable models.

1. **Specifying Problems**

2. **The RISC Algorithm Language (RISCAL)**

3. **Modeling Computations**

Specifying Problems

- A (computational) problem:

  Input: $x_1 \in T_1, \ldots, x_n \in T_n$ where I_x

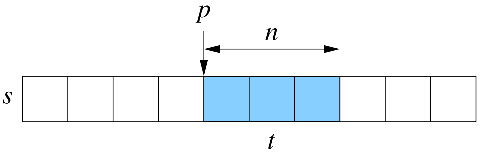
  Output: $y_1 \in U_1, \ldots, y_m \in U_m$ where $O_{x,y}$

- Input variables $x_1, \ldots, x_n$.
  - With types $T_1, \ldots, T_n$.
- Input condition (precondition) I_x.
  - A formula whose free variables occur in $x_1, \ldots, x_n$.
- Output variables $y_1, \ldots, y_m$.
  - With types $U_1, \ldots, U_m$.
- Output condition (postcondition) $O_{x,y}$.
  - A formula whose free variables occur in $x_1, \ldots, x_n, y_1, \ldots, y_m$.

Formulas refer to functions and predicates that characterize the problem domain.

Example

Extract from a finite sequence s a subsequence of length n starting at position p.



Input: $s \in T^*, n \in \mathbb{N}, p \in \mathbb{N}$ where

$$n + p \leq \mathrm{length}(s)$$

Output: $t \in T^*$ where

$$\mathrm{length}(t) = n \wedge$$
$$\forall i \in \mathbb{N}.\ i < n \Rightarrow t[i] = s[i + p]$$

The resulting sequence must have appropriate length and contents.

Implementing Problem Specifications

- The specification demands a function $f : T_1 \times \ldots \times T_n \rightarrow U_1 \times \ldots \times U_m$ such that

$$\forall x_1 \in T_1, \ldots, x_n \in T_n.\ I_x \Rightarrow \text{ let } (y_1, \ldots, y_m) = f(x_1, \ldots, x_n) \text{ in } O_{x,y}$$

  - □ For all arguments $x_1, \ldots, x_n$ that satisfy the input condition,
  - □ the result $(y_1, \ldots, y_m)$ of f satisfies the output condition.
- The specification itself already implicitly defines such a function:

$$f(x_1, \ldots, x_n) := \text{choose } y_1 \in U_1, \ldots, y_m \in U_m.\ O_{x,y}$$

  - □ An implicit function definition (whose result is arbitrary, if no values satisfy O).
- An actual implementation must provide an explicitly defined function.
  - □ Right-side of definition is a term that describes a constructive computation.

The ultimate goal of computer science/mathematics is to provide explicit definitions of functions (i.e., programs) that implement problem specifications.

Function Definitions

- An (explicit) function definition

$$f : T_1 \times \ldots \times T_n \to T$$
$$f(x_1, \ldots, x_n) := t_x$$

  - Special case $n = 0$: a constant definition $c : T, c := t$.
- Function constant f of arity n.
- Type signature $T_1 \times \ldots \times T_n \to T$.
- Parameters $x_1, \ldots, x_n$ (variables).
- Body t_x (a term whose free variables occur in $x_1, \ldots, x_n$).

We thus know $\forall x_1 \in T_1, \ldots, x_n \in T_n.\ f(x_1, \ldots, x_n) = t_x$.

Examples

- Definition: Let x and y be natural numbers. Then the square sum of x and y is the sum of the squares of x and y.

$$\begin{aligned}&\mathrm{squaresum}\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}\\ &\mathrm{squaresum}(x,y) := x^2 + y^2\end{aligned}$$

- Definition: Let x and y be natural numbers. Then the squared sum of x and y is the square of z where z is the sum of x and y.

$$\begin{aligned}&\mathrm{sumsquared}\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}\\ &\mathrm{sumsquared}(x,y) := \textsf{let } z = x + y \textsf{ in } z^2\end{aligned}$$

- Definition: Let n be a natural number. Then the square sum set of n is the set of the square sums of all numbers x and y from 1 to n.

$$\begin{aligned}&\mathrm{squaresumset}\colon \mathbb{N}\to\mathcal{P}(\mathbb{N})\\ &\mathrm{squaresumset}(n) := \{\mathrm{squaresum}(x,y) \mid x,y\in\mathbb{N}\wedge 1\le x\le n\wedge 1\le y\le n\}\end{aligned}$$

Predicate Definitions

- An (explicit) predicate definition

$$p \subseteq T_1 \times \ldots \times T_n$$
$$p(x_1, \ldots, x_n) :\Leftrightarrow F_x$$

- Predicate constant p of arity n.
- Type signature $T_1 \times \ldots \times T_n$.
- Parameters $x_1, \ldots, x_n$ (variables).
- Body F_x (a formula whose free variables occur in $x_1, \ldots, x_n$).

We thus know $\forall x_1 \in T_1, \ldots, x_n \in T_n.\ p(x_1, \ldots, x_n) \Leftrightarrow F_x$.

Examples

- Definition: Let x, y be natural numbers. Then x divides y (written as $x|y$) if $x \cdot z = y$ for some natural number z.

$$\begin{aligned}&\text{␣}|\text{␣} \subseteq \mathbb{N} \times \mathbb{N}\\&x|y :\Leftrightarrow \exists z \in \mathbb{N}.\ x \cdot z = y\end{aligned}$$

- Definition: Let x be a natural number. Then x is prime if x is at least two and the only divisors of x are one and x itself.

$$\begin{aligned}&\mathrm{isprime} \subseteq \mathbb{N}\\&\mathrm{isprime}(x) :\Leftrightarrow x \geq 2 \land \forall y \in \mathbb{N}.\ y|x \Rightarrow y = 1 \lor y = x\end{aligned}$$

- Definition: Let p, n be a natural numbers. Then p is a prime factor of n, if p is prime and divides n.

$$\begin{aligned}&\mathrm{isprimefactor} \subseteq \mathbb{N} \times \mathbb{N}\\&\mathrm{isprimefactor}(p, n) :\Leftrightarrow \mathrm{isprime}(p) \land p|n\end{aligned}$$

Implicit Definitions

- An implicit function definition

$$f : T_1 \times \ldots \times T_n \to T$$
$$f(x_1, \ldots, x_n) := \text{choose } y \in T.\ F_{x,y}$$

- Function constant f of arity n.
- Type signature $T_1 \times \ldots \times T_n \to T$.
- Parameters $x_1, \ldots, x_n$ (variables).
- Result variable y.
- Result condition $F_{x,y}$ (a formula whose free variables occur in $x_1, \ldots, x_n, y$).

We thus know $\forall x_1 \in T_1, \ldots, x_n \in T_n.\ (\exists y \in T.\ F_{x,y}) \Rightarrow$ let $y = f(x_1, \ldots, x_n)$ in $F_{x,y}$.

Examples

- Definition: A root of x is some y such that y squared is x (if such a y exists).

$$\begin{aligned} &\mathrm{aRoot}\colon \mathbb{R} \to \mathbb{R} \\ &\mathrm{aRoot}(x) := \text{ choose } y \in \mathbb{R}.\ y^2 = x \end{aligned}$$

- Definition: The root of $x \geq 0$ is that y such that the square of y is x and $y \geq 0$.

$$\begin{aligned} &\mathrm{theRoot}\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \\ &\mathrm{theRoot}(x) := \text{choose } y \in \mathbb{R}_{\geq 0}.\ y^2 = x \wedge y \geq 0 \end{aligned}$$

- Definition: The quotient q of m and $n \neq 0$ is such that $m = n \cdot q + r$ for some $r < n$.

$$\begin{aligned} &\mathrm{quotient}\colon \mathbb{N} \times \mathbb{N} \backslash \{0\} \to \mathbb{N} \\ &\mathrm{quotient}(m, n) := \text{choose } q \in \mathbb{N}.\ \exists r \in \mathbb{N}.\ m = n \cdot q + r \wedge r < n \end{aligned}$$

- Definition: The $gcd(x, y)$ of x, y (not both 0), is the greatest number dividing x and y.

$$\begin{aligned} &\mathrm{gcd}\colon (\mathbb{N} \times \mathbb{N}) \backslash \{(0,0)\} \to \mathbb{N} \\ &\mathrm{gcd}(x, y) := \text{choose } z \in \mathbb{N}.\ z|x \wedge z|y \wedge \forall z' \in \mathbb{N}.\ z'|x \wedge z'|y \Rightarrow z' \leq z \end{aligned}$$

Function result need not be uniquely defined (may be even arbitrary).

Predicates versus Functions

A predicate gives rise to functions in two ways.

- A predicate:

$$\mathrm{isprimefactor} \subseteq \mathbb{N} \times \mathbb{N}$$
$$\mathrm{isprimefactor}(p, n) :\Leftrightarrow \mathrm{isprime}(p) \wedge p | n$$

- An implicitly defined function:

$$\mathrm{someprimefactor}\colon \mathbb{N} \to \mathbb{N}$$
$$\mathrm{someprimefactor}(n) := \text{ choose } p \in \mathbb{N}.\ \mathrm{isprimefactor}(p, n)$$

- An explicitly defined function whose result is a set:

$$\mathrm{allprimefactors}\colon \mathbb{N} \to \mathcal{P}(\mathbb{N})$$
$$\mathrm{allprimefactors}(n) := \{p \mid p \in \mathbb{N} \wedge \mathrm{isprimefactor}(p, n)\}$$

The preferred style of definition is a matter of taste and purpose.

The Adequacy of Specifications

Given a specification

Input: x where P_x **Output:** y where $Q_{x,y}$

we may ask the following questions:

- Is precondition satisfiable? $(\exists x.\ P_x)$
  - Otherwise no input is allowed.
- Is precondition not trivial? $(\exists x.\ \neg P_x)$
  - Otherwise every input is allowed, why then the precondition?
- Is postcondition always satisfiable? $(\forall x.\ P_x \Rightarrow \exists y. Q_{x,y})$
  - Otherwise no implementation is legal.
- Is postcondition not always trivial? $(\exists x, y.\ P_x \wedge \neg Q_{x,y})$
  - Otherwise every implementation is legal.
- Is result unique? $(\forall x, y_1, y_2.\ P_x \wedge Q_{x,y_1} \wedge Q_{x,y_2} \Rightarrow y_1 = y_2)$
  - Whether this is required, depends on our expectations.

Example: The Problem of Integer Division

Input: $m \in \mathbb{N}, n \in \mathbb{N}$ **Output:** $q \in \mathbb{N}, r \in \mathbb{N}$ where $m = n \cdot q + r$

- The postcondition is always satisfiable but not trivial.
  - For $m = 13, n = 5$, e.g., $q = 2, r = 3$ is legal but $q = 2, r = 4$ is not.
- But the result is not unique.
  - For $m = 13, n = 5$, both $q = 2, r = 3$ and $q = 1, r = 8$ are legal.

Input: $m \in \mathbb{N}, n \in \mathbb{N}$ **Output:** $q \in \mathbb{N}, r \in \mathbb{N}$ where $m = n \cdot q + r \wedge r < n$

- Now the postcondition is not always satisfiable.
  - For $m = 13, n = 0$, no output is legal.

Input: $m \in \mathbb{N}, n \in \mathbb{N}$ where $n \neq 0$ **Output:** $q \in \mathbb{N}, r \in \mathbb{N}$ where $m = n \cdot q + r \wedge r < n$

- The precondition is not trival but satisfiable.
  - $m = 13, n = 0$ is not legal but $m = 13, n = 5$ is.
- The postcondition is always satisfiable and result is unique.
  - For $m = 13, n = 5$, only $q = 2, r = 3$ is legal.

Example: The Problem of Linear Search

Given a finite integer sequence a and an integer x, determine the smallest position p at which x occurs in a ($p = -1$, if x does not occur in a).

Example: $a = [2, 3, 5, 7, 5, 11], x = 5 \rightsquigarrow p = 2$

Input: $a \in \mathbb{Z}^*, x \in \mathbb{Z}$

Output: $p \in \mathbb{N} \cup \{-1\}$ where

```
let $n = \mathrm{length}(a)$ in
if $\exists p \in \mathbb{N}.\ \underline{p < n \land a[p] = x}$
   then $\underline{p < n \land a[p] = x} \land \left(\forall q \in \mathbb{N}.\ \underline{q < n \land a[q] = x} \Rightarrow p \leq q\right)$
   else $p = -1$
```

All inputs are legal; a result with the specified property always exists and is uniquely determined.

Example: The Problem of Binary Search

Given a finite integer sequence a sorted in ascending order and an integer x, determine some position p at which x occurs in a ($p = -1$, if x does not occur in a).

Example: $a = [2, 3, 5, 5, 5, 7, 11], x = 5 \rightsquigarrow p \in \{2, 3, 4\}$

Input: $a \in \mathbb{Z}^*, x \in \mathbb{Z}$ where

let $n = \text{length}(a)$ in $\forall k \in \mathbb{N}.\ k < n - 1 \Rightarrow a[k] \leq a[k+1]$

Output: $p \in \mathbb{N} \cup \{-1\}$ where

if $\exists p \in \mathbb{N}.\ \underline{p < n \wedge a[p] = x}$

then $\underline{p < n \wedge a[p] = x}$

else $p = -1$

Not all inputs are legal; for every legal input, a result with the specified property exists but may not be unique.

Example: The Problem of Sorting

Given a finite integer sequence a, determine that permutation b of a that is sorted in ascending order.

Example: $a = [5, 3, 7, 2, 3] \rightsquigarrow b = [2, 3, 3, 5, 7]$

Input: $a \in \mathbb{Z}^*$

Output: $b \in \mathbb{Z}^*$**where**

let $n = \mathrm{length}(a)$ in

$\mathrm{length}(b) = n \wedge (\forall k \in \mathbb{N}.\ k < n - 1 \Rightarrow b[k] \leq b[k+1])\ \wedge$

$\exists p \in \mathbb{N}^*.\ \mathrm{length}(p) = n \wedge$

$(\forall k \in \mathbb{N}.\ k < n \Rightarrow p[k] < n) \wedge$

$(\forall k1 \in \mathbb{N}, k2 \in \mathbb{N}.\ k1 < n \wedge k2 < n \wedge k1 \neq k2 \Rightarrow p[k1] \neq p[k2]) \wedge$

$(\forall k \in \mathbb{N}.k < n \Rightarrow a[k] = b[p[k]])$

All inputs are legal; the specified result exists and is uniquely determined.

1. Specifying Problems

2. **The RISC Algorithm Language (RISCAL)**

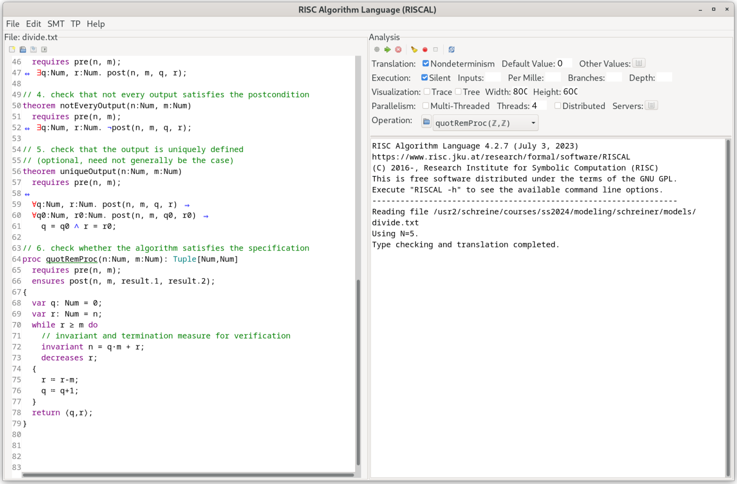
3. Modeling Computations

The RISC Algorithm Language (RISCAL)

- A system for formally modeling mathematical theories and algorithms.
  - Research Institute for Symbolic Computation (RISC), 2016–.
    - `http://www.risc.jku.at/research/formal/software/RISCAL`
  - Implemented in Java with SWT library for the GUI.
    - Tested under Linux only; freely available as open source (GPL3).
- A language for the defining mathematical theories and algorithms.
  - A static type system with only finite types (of parameterized sizes).
  - Predicates, explicitly (also recursively) and implicitly def.d functions.
  - Theorems (universally quantified predicates expected to be true).
  - Procedures (also recursively defined).
  - Pre- and post-conditions, invariants, termination measures.
- A framework for evaluating/executing all definitions.
  - Model checking: predicates, functions, theorems, procedures, annotations may be evaluated/executed for all possible inputs.
  - All paths of a non-deterministic execution may be elaborated.
  - The execution/evaluation may be visualized.

The RISC Algorithm Language (RISCAL)

```
RISCAL divide.txt &
```



```
  requires pre(n, m);
⇔ ∃q:Num, r:Num. post(n, m, q, r);

// 4. check that not every output satisfies the postcondition
theorem notEveryOutput(n:Num, m:Num)
  requires pre(n, m);
⇔ ∃q:Num, r:Num. ¬post(n, m, q, r);

// 5. check that the output is uniquely defined
// (optional, need not generally be the case)
theorem uniqueOutput(n:Num, m:Num)
  requires pre(n, m);
⇔
  ∀q:Num, r:Num. post(n, m, q, r) →
  ∀q0:Num, r0:Num. post(n, m, q0, r0) →
    q = q0 ∧ r = r0;

// 6. check whether the algorithm satisfies the specification
proc quotRemProc(n:Num, m:Num): Tuple[Num,Num]
  requires pre(n, m);
  ensures post(n, m, result.1, result.2);
{
  var q: Num = 0;
  var r: Num = n;
  while r ≥ m do
    // invariant and termination measure for verification
    invariant n = q·m + r;
    decreases r;
  {
    r ≔ r-m;
    q ≔ q+1;
  }
  return ⟨q,r⟩;
}
```

Translation: ☑ Nondeterminism Default Value: 0 Other Values:

Execution: ☑ Silent Inputs: Per Mille: Branches: Depth:

Visualization: □ Trace □ Tree Width: 800 Height: 600

Parallelism: □ Multi-Threaded Threads: 4 □ Distributed Servers:

Operation: quotRemProc(ℤ,ℤ)

```
RISC Algorithm Language 4.2.7 (July 3, 2023)
https://www.risc.jku.at/research/formal/software/RISCAL
(C) 2016-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCAL -h" to see the available command line options.
------------------------------------------------------------------
Reading file /usr2/schreine/courses/ss2024/modeling/schreiner/models/
divide.txt
Using N=5.
Type checking and translation completed.
```

Using RISCAL

See also the (printed/online) “Tutorial and Reference Manual”.

- Press button 💾 (or <Ctrl>-s) to save specification.
  - Automatically processes (parses and type-checks) specification.
  - Press button ⚙ to re-process specification.
- Choose values for undefined constants in specification.
  - Natural number for `val const: ℕ`.
  - Default Value: used if no other value is specified.
  - Other Values: specific values for individual constants.
- Select Operation from menu and then press button ➡.
  - Executes operation for chosen constant values and all possible inputs.
  - Option Silent: result of operation is not printed.
  - Option Nondeterminism: all execution paths are taken.
  - Option Multi-threaded: multiple threads execute different inputs.
  - Press buttton ✖ to abort execution.

During evaluation all annotations (pre/postconditions, etc.) are checked.

Typing Mathematical Symbols

ASCII String	Unicode Character
Int	ℤ
Nat	ℕ
:=	≔
true	⊤
false	⊥
~	¬
/\	∧
\/	∨
=>	⇒
<=>	⇔
forall	∀
exists	∃
sum	∑
product	∏

ASCII String	Unicode Character
~=	≠
<=	≤
>=	≥
*	·
times	×
{}	∅
intersect	∩
union	∪
Intersect	⋂
Union	⋃
isin	∈
subseteq	⊆
<<	⟨
>>	⟩

Type the ASCII string and press <Ctrl>-# to get the Unicode character.

Example: Quotient and Remainder

Given naturals n and m, compute the quotient q and remainder r of n divided by m.

```
// the type of natural numbers less than equal N
val N: ℕ;
type Num = ℕ[N];

// the precondition of the computation
pred pre(n:Num, m:Num) ⇔ m ≠ 0;

// the postcondition, first formulation
pred post1(n:Num, m:Num, q:Num, r:Num) ⇔
  n = m·q + r ∧
  ∀q0:Num, r0:Num.
    n = m·q0 + r0 ⇒ r ≤ r0;

// the postcondition, second formulation
pred post2(n:Num, m:Num, q:Num, r:Num) ⇔
  n = m·q + r ∧ r < m;
```

We will investigate this specification.

Example: Quotient and Remainder

```
// for all inputs that satisfy the precondition
// both formulations are equivalent:
// ∀n:Num, m:Num, q:Num, r:Num.
//   pre(n, m) ⇒ (post1(n, m, q, r) ⇔ post2(n, m, q, r));
theorem postEquiv(n:Num, m:Num, q:Num, r:Num)
  requires pre(n, m);
⇔ post1(n, m, q, r) ⇔ post2(n, m, q, r);

// we will thus use the simpler formulation from now on
pred post(n:Num, m:Num, q:Num, r:Num) ⇔ post2(n, m, q, r);
```

Check equivalence for all values that satisfy the precondition.

Example: Quotient and Remainder

Choose e.g. $N = 5$.

- Switch option Silent off:

```
Executing postEquiv(ℤ,ℤ,ℤ,ℤ) with all 1296 inputs.
Ignoring inadmissible inputs...
Run 6 of deterministic function postEquiv(0,1,0,0):
Result (0 ms): true
Run 7 of deterministic function postEquiv(1,1,0,0):
Result (0 ms): true
...
Run 1295 of deterministic function postEquiv(5,5,5,5):
Result (0 ms): true
Execution completed for ALL inputs (6314 ms, 1080 checked, 216 inadmissible).
```

- Switch option Silent on:

```
Executing postEquiv(ℤ,ℤ,ℤ,ℤ) with all 1296 inputs.
Execution completed for ALL inputs (244 ms, 1080 checked, 216 inadmissible).
```

If theorem is false for some input, an error message is displayed.

Example: Quotient and Remainder

Drop precondition from theorem.

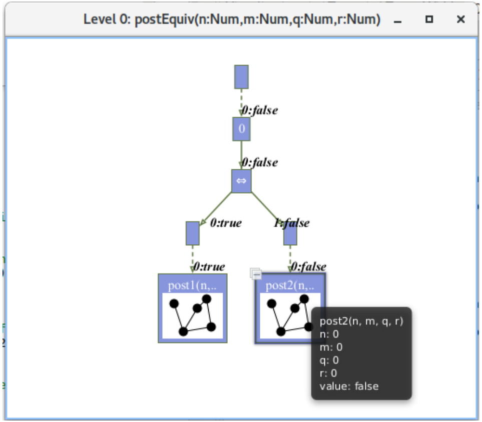
```
theorem postEquiv(n:Num, m:Num, q:Num, r:Num) ⇔
  // requires pre(n, m);
  post1(n, m, q, r) ⇔ post2(n, m, q, r);
```

```
Executing postEquiv(ℤ,ℤ,ℤ,ℤ) with all 1296 inputs.
Run 0 of deterministic function postEquiv(0,0,0,0):
ERROR in execution of postEquiv(0,0,0,0): evaluation of
  postEquiv
at line 25 in file divide.txt:
  theorem is not true
ERROR encountered in execution.
```

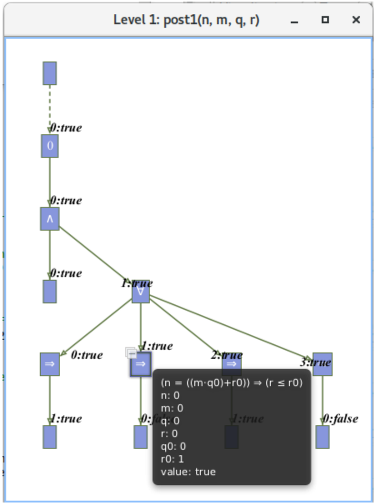
For $n = 0, m = 0, q = 0, r = 0$, the modified theorem is not true.

Visualizing the Formula Evaluation

Select $N = 1$ and visualization option "Tree".



```
post2(n, m, q, r)
n: 0
m: 0
q: 0
r: 0
value: false
```



```
(n = ((m·q0)+r0)) ⇒ (r ≤ r0)
n: 0
m: 0
q: 0
r: 0
q0: 0
r0: 1
value: true
```

Investigate the (pruned) evaluation tree to determine how the truth value of a formula was derived (double click to zoom into/out of predicates).

Example: Quotient and Remainder

Switch option “Nondeterminism” on.

```
// 1. investigate whether the specified input/output combinations are as desired
fun quotremFun(n:Num, m:Num): Tuple[Num,Num]
  requires pre(n, m);
  ensures post(n, m, result.1, result.2);
= choose q:Num, r:Num with post(n, m, q, r);

Executing quotremFun(ℤ,ℤ) with all 36 inputs.
Ignoring inadmissible inputs...
Branch 0:6 of nondeterministic function quotremFun(0,1):
Result (0 ms): [0,0]
...
Branch 1:35 of nondeterministic function quotremFun(5,5):
No more results (14 ms).
Execution completed for ALL inputs (413 ms, 30 checked, 6 inadmissible).
```

First validation by inspecting the values determined by output condition (nondeterminism may produce for some inputs multiple outputs).

Example: Quotient and Remainder

```
// 2. check that some but not all inputs are allowed
theorem someInput() ⇔ ∃n:Num, m:Num. pre(n, m);
theorem notEveryInput() ⇔ ∃n:Num, m:Num. ¬pre(n, m);

Executing someInput().
Execution completed (0 ms).
Executing notEveryInput().
Execution completed (0 ms).
```

A very rough validation of the input condition.

Example: Quotient and Remainder

```
// 3. check whether for all inputs that satisfy the precondition
// there are some outputs that satisfy the postcondition
theorem someOutput(n:Num, m:Num)
  requires pre(n, m);
⇔ ∃q:Num, r:Num. post(n, m, q, r);

// 4. check that not every output satisfies the postcondition
theorem notEveryOutput(n:Num, m:Num)
  requires pre(n, m);
⇔ ∃q:Num, r:Num. ¬post(n, m, q, r);
```

```
Executing someOutput(ℤ,ℤ) with all 36 inputs.
Execution completed for ALL inputs (5 ms, 30 checked, 6 inadmissible).
Executing notEveryOutput(ℤ,ℤ) with all 36 inputs.
Execution completed for ALL inputs (5 ms, 30 checked, 6 inadmissible).
```

A very rough validation of the output condition.

Example: Quotient and Remainder

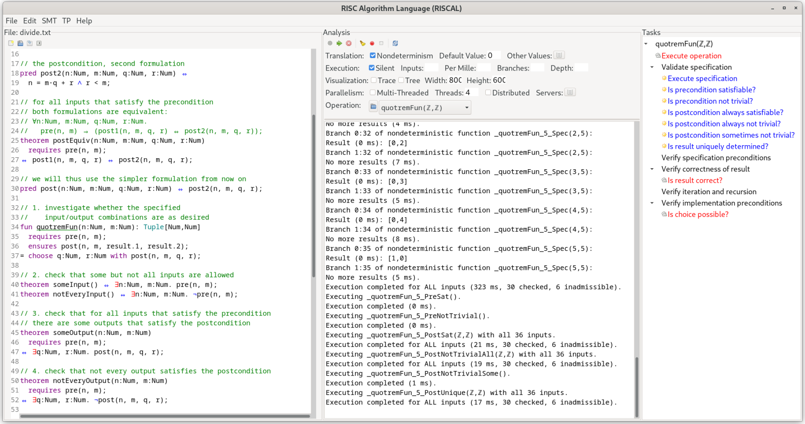
```
// 5. check that the output is uniquely defined
// (optional, need not generally be the case)
theorem uniqueOutput(n:Num, m:Num)
  requires pre(n, m);
⇔
  ∀q:Num, r:Num. post(n, m, q, r) ⇒
  ∀q0:Num, r0:Num. post(n, m, q0, r0) ⇒
    q = q0 ∧ r = r0;
```

```
Executing uniqueOutput(ℤ,ℤ) with all 36 inputs.
Execution completed for ALL inputs (18 ms, 30 checked, 6 inadmissible).
```

The output condition indeed determines the outputs uniquely.

Validating the Specification of an Operation

Select operation `quotRemFun` and press the button “Show/Hide Tasks”.



```

// the postcondition, second formulation
pred post2(n:Num, m:Num, q:Num, r:Num) ⇔
  n = m·q + r ∧ r < m;

// for all inputs that satisfy the precondition
// both formulations are equivalent:
// ∀n:Num, m:Num, q:Num, r:Num.
//   pre(n, m) ⇒ (post1(n, m, q, r) ⇔ post2(n, m, q, r));
theorem postEquiv(n:Num, m:Num, q:Num, r:Num)
  requires pre(n, m);
⇔ post1(n, m, q, r) ⇔ post2(n, m, q, r);

// we will thus use the simpler formulation from now on
pred post(n:Num, m:Num, q:Num, r:Num) ⇔ post2(n, m, q, r);

// 1. investigate whether the specified
//    input/output combinations are as desired
fun quotremFun(n:Num, m:Num): Tuple[Num,Num]
  requires pre(n, m);
  ensures post(n, m, result.1, result.2);
= choose q:Num, r:Num with post(n, m, q, r);

// 2. check that some but not all inputs are allowed
theorem someInput() ⇔ ∃n:Num, m:Num. pre(n, m);
theorem notEveryInput() ⇔ ∃n:Num, m:Num. ¬pre(n, m);

// 3. check that for all inputs that satisfy the precondition
// there are some outputs that satisfy the postcondition
theorem someOutput(n:Num, m:Num)
  requires pre(n, m);
⇔ ∃q:Num, r:Num. post(n, m, q, r);

// 4. check that not every output satisfies the postcondition
theorem notEveryOutput(n:Num, m:Num)
  requires pre(n, m);
⇔ ∃q:Num, r:Num. ¬post(n, m, q, r);

```

Translation: ☑ Nondeterminism Default Value: 0 Other Values:

Execution: ☑ Silent Inputs: Per Mille: Branches: Depth:

Visualization: □ Trace □ Tree Width: 80(Height: 60(

Parallelism: □ Multi-Threaded Threads: 4 □ Distributed Servers:

Operation: quotremFun(ℤ,ℤ)

```
No more results (4 ms).
Branch 0:32 of nondeterministic function _quotremFun_5_Spec(2,5):
Result (0 ms): [0,2]
Branch 1:32 of nondeterministic function _quotremFun_5_Spec(2,5):
No more results (7 ms).
Branch 0:33 of nondeterministic function _quotremFun_5_Spec(3,5):
Result (0 ms): [0,3]
Branch 1:33 of nondeterministic function _quotremFun_5_Spec(3,5):
No more results (5 ms).
Branch 0:34 of nondeterministic function _quotremFun_5_Spec(4,5):
Result (0 ms): [0,4]
Branch 1:34 of nondeterministic function _quotremFun_5_Spec(4,5):
No more results (8 ms).
Branch 0:35 of nondeterministic function _quotremFun_5_Spec(5,5):
Result (0 ms): [1,0]
Branch 1:35 of nondeterministic function _quotremFun_5_Spec(5,5):
No more results (5 ms).
Execution completed for ALL inputs (323 ms, 30 checked, 6 inadmissible).
Executing _quotremFun_5_PreSat().
Execution completed (0 ms).
Executing _quotremFun_5_PreNotTrivial().
Execution completed (0 ms).
Executing _quotremFun_5_PostSat(ℤ,ℤ) with all 36 inputs.
Execution completed for ALL inputs (21 ms, 30 checked, 6 inadmissible).
Executing _quotremFun_5_PostNotTrivialAll(ℤ,ℤ) with all 36 inputs.
Execution completed for ALL inputs (19 ms, 30 checked, 6 inadmissible).
Executing _quotremFun_5_PostNotTrivialSome().
Execution completed (1 ms).
Executing _quotremFun_5_PostUnique(ℤ,ℤ) with all 36 inputs.
Execution completed for ALL inputs (17 ms, 30 checked, 6 inadmissible).
```

- quotremFun(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?

Automatic generation of those formulas that validate a specification.

Example: Quotient and Remainder

Right-click to print definition of a formula, double-click to check it.

```
For every input, is postcondition true for only one output?

theorem _quotremFun_5_PostUnique(n:Num, m:Num)
requires pre(n, m);
 ⇔ ∀result:Tuple[Num,Num] with post(n, m, result.1, result.2).
   (∀_result:Tuple[Num,Num] with let result = _result in
     post(n, m, result.1, result.2). (result = _result));

Using N=5.
Type checking and translation completed.
Executing _quotremFun_5_PostUnique(ℤ,ℤ) with all 36 inputs.
Execution completed for ALL inputs (7 ms, 30 checked, 6 inadmissible).
```

The output is indeed uniquely defined by the output condition.

Example: Quotient and Remainder

```
// 6. check whether the algorithm satisfies the specification
proc quotRemProc(n:Num, m:Num): Tuple[Num,Num]
  requires pre(n, m);
  ensures let q=result.1, r=result.2 in post(n, m, q, r);
{
  var q: Num = 0;
  var r: Num = n;
  while r ≥ m do
  {
    r := r-m;
    q := q+1;
  }
  return ⟨q,r⟩;
}
```

Check whether the algorithm satisfies the specification.

Example: Quotient and Remainder

```
Executing quotRemProc(ℤ,ℤ) with all 36 inputs.
Ignoring inadmissible inputs...
Run 6 of deterministic function quotRemProc(0,1):
Result (0 ms): [0,0]
Run 7 of deterministic function quotRemProc(1,1):
Result (0 ms): [1,0]
...
Run 32 of deterministic function quotRemProc(2,5):
Result (0 ms): [0,2]
Run 33 of deterministic function quotRemProc(3,5):
Result (0 ms): [0,3]
Run 34 of deterministic function quotRemProc(4,5):
Result (0 ms): [0,4]
Run 35 of deterministic function quotRemProc(5,5):
Result (1 ms): [1,0]
Execution completed for ALL inputs (161 ms, 30 checked, 6 inadmissible).
```

A verification of the algorithm by checking all possible executions.

Example: Quotient and Remainder

```
proc quotRemProc(n:Num, m:Num): Tuple[Num,Num]
  requires pre(n, m);
  ensures post(n, m, result.1, result.2);
{
  var q: Num = 0; var r: Num = n;
  while r > m do // error!
  {
    r := r-m; q := q+1;
  }
  return ⟨q,r⟩;
}
```

```
Executing quotRemProc(ℤ,ℤ) with all 36 inputs.
ERROR in execution of quotRemProc(1,1): evaluation of
  ensures let q = result.1, r = result.2 in post(n, m, q, r);
at line 65 in file divide.txt:
  postcondition is violated by result [0,1]
ERROR encountered in execution.
```

A falsificaton of an incorrect algorithm.

Example: Sorting an Array

```
val N:Nat; val M:Nat;
type nat = Nat[M]; type array = Array[N,nat]; type index = Nat[N-1];

proc sort(a:array): array
  ensures ∀i:nat. i < N-1 ⇒ result[i] ≤ result[i+1];
  ensures ∃p:Array[N,index]. (∀i:index,j:index. i ≠ j ⇒ p[i] ≠ p[j]) ∧
                             (∀i:index. a[i] = result[p[i]]);
{
  var b:array = a;
  for var i:Nat[N]:=1; i<N; i:=i+1 do {
    var x:nat := b[i];
    var j:Int[-1,N] := i-1;
    while j ≥ 0 ∧ b[j] > x do {
      b[j+1] := b[j];
      j := j-1;
    }
    b[j+1] := x;
  }
  return b;
}
```

Example: Sorting an Array

```
Using N=5.
Using M=5.
Type checking and translation completed.
Executing sort(Array[ℤ]) with all 7776 inputs.
1223 inputs (1223 checked, 0 inadmissible, 0 ignored)...
2026 inputs (2026 checked, 0 inadmissible, 0 ignored)...
...
5792 inputs (5792 checked, 0 inadmissible, 0 ignored)...
6118 inputs (6118 checked, 0 inadmissible, 0 ignored)...
6500 inputs (6500 checked, 0 inadmissible, 0 ignored)...
6788 inputs (6788 checked, 0 inadmissible, 0 ignored)...
7070 inputs (7070 checked, 0 inadmissible, 0 ignored)...
7354 inputs (7354 checked, 0 inadmissible, 0 ignored)...
7634 inputs (7634 checked, 0 inadmissible, 0 ignored)...
Execution completed for ALL inputs (32606 ms, 7776 checked, 0 inadmissible).
Not all nondeterministic branches may have been considered.
```

Also this algorithm can be automatically checked.

Model Checking versus Proving

Two fundamental techniques for validation/verification.

- Model checking: processing a semantic model.
  - Fully automatic, no human interaction is required.
  - Completely possible only if the model is finite.
  - State space explosion: “finite” actually means “not too big”.
- Proving: constructing a logical deduction.
  - Assumes a sound deduction calculus.
  - Also possible if the model is infinite.
  - Complexity of deduction is independent of size of model.
  - Many properties can be automatically proved (automated reasoners); in general, however, interaction with a human is required (proof assistants).

While verifying the validity of a conjecture generally requires deduction, its invalidity can be often quickly established by checking.

1. Specifying Problems

2. The RISC Algorithm Language (RISCAL)

3. **Modeling Computations**

Computational Systems

Programs are just special cases of “(computational) systems”.

- Computational System
  - One or more active components.
  - Deterministic or nondeterministic behavior.
  - May or may not terminate.
- Safety
  - “Nothing bad will ever happen.”
  - Partial correctness of programs: for every admissible input, if the program terminates, its output does not violate the output condition.
- Liveness
  - “Something good will eventually happen.”
  - Termination of programs: for every input, the program eventually terminates.

General goal is to establish the safety and liveness of computational systems.

Transition Systems

Any computational system can be modelled as a transition system $T = (S, I, R)$.

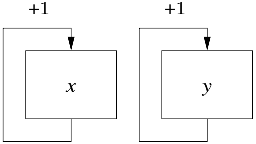
- State space $S = S_1 \times \ldots \times S_n$: the set of all possible system states.
  - □ Determined by the possible values of system variables $x_1, \ldots, x_n$ with values from (finite or infinite) domains $S_1, \ldots, S_n$.
- Initial states $I \subseteq S$: the possible starts of the execution of the system.
  - □ Typically defined by an a predicate I_x on the system variables $x_1, \ldots, x_n$.
- Transition relation $R \subseteq S \times S$: the possible execution steps.
  - □ Typically defined by a predicate $R_{x,x'}$ between the prestate values x and the poststate values x' of the program variables.

Nondeterminism: for some prestate x there may be multiple poststates x'.

Example

System $C = (S, I, R)$ with counters x und y which may be independently incremented.

$$S := \mathbb{Z} \times \mathbb{Z}$$
$$I(x, y) :\Leftrightarrow x = y \wedge y \geq 0$$
$$R(\langle x, y\rangle, \langle x', y'\rangle) :\Leftrightarrow$$
$$(x' = x + 1 \wedge y' = y) \vee$$
$$(x' = x \wedge y' = y + 1)$$



- Infinitely many starting states.

$$[x = 0, y = 0], [x = 1, y = 1], [x = 2, y = 2], \ldots$$

- In each state two possibilities.

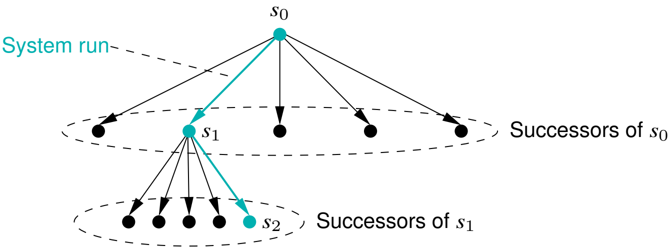
$$\begin{aligned}[x = 2, y = 3] &\rightarrow [x = 3, y = 3]\\ &\rightarrow [x = 2, y = 4]\end{aligned}$$

A nondeterministic system.

System Runs

Transition system $T = (S, I, R)$.

- System run: (finite or infinite) sequence $s_0 \to s_1 \to s_2 \to \ldots$ of states in S.
  - s_0 is initial: $I(s_0)$.
  - $s_i \to s_{i+1}$ ist a transition: $R(s_0, s_1)$.
  - If run stops in s_n, then s_n has no successor: $\neg R(s_n, s')$, for all $s' \in S$.



System runs can be understood as paths in a directed graph.

Example

System $C = (S, I, R)$.

$$
\begin{aligned}
&S := \mathbb{Z} \times \mathbb{Z} \\
&I(x, y) :\Leftrightarrow x = y \wedge y \geq 0 \\
&R(\langle x, y\rangle, \langle x', y'\rangle) :\Leftrightarrow \\
&\quad (x' = x + 1 \wedge y' = y) \vee \\
&\quad (x' = x \wedge y' = y + 1)
\end{aligned}
$$

- Safety: $\Box(x \geq 0 \wedge y \geq 0)$
  - Both x als y never become negative.
  - True, because every system run has this property.
- Liveness: $\Diamond x \geq 1$.
  - Variable x eventually becomes greater equal 1.
  - False, because this system run does not have this property.

$$[x = 0, y = 0] \rightarrow [x = 0, y = 1] \rightarrow [x = 0, y = 2] \rightarrow [x = 0, y = 3] \rightarrow \ldots$$

For establishing liveness properties, “unfair” system runs must be ruled out.

Verifying Safety

- $M \models \Box F$.
  - Verify that formula F is an invariant of system M.
- $M = (S, I, R)$.
  - $I(s) :\Leftrightarrow \ldots$
  - $R(s, s') :\Leftrightarrow R_0(s, s') \vee R_1(s, s') \vee \ldots \vee R_{n-1}(s, s')$.
- Proof by induction.
  - $\forall s.\ I(s) \Rightarrow F(s)$.
    - F holds in every initial state.
  - $\forall s, s'.\ F(s) \wedge R(s, s') \Rightarrow F(s')$.
    - Each transition preserves F.
    - Reduces to a number of subproofs:

$$F(s) \wedge R_0(s, s') \Rightarrow F(s')$$

$$\ldots$$

$$F(s) \wedge R_{n-1}(s, s') \Rightarrow F(s')$$

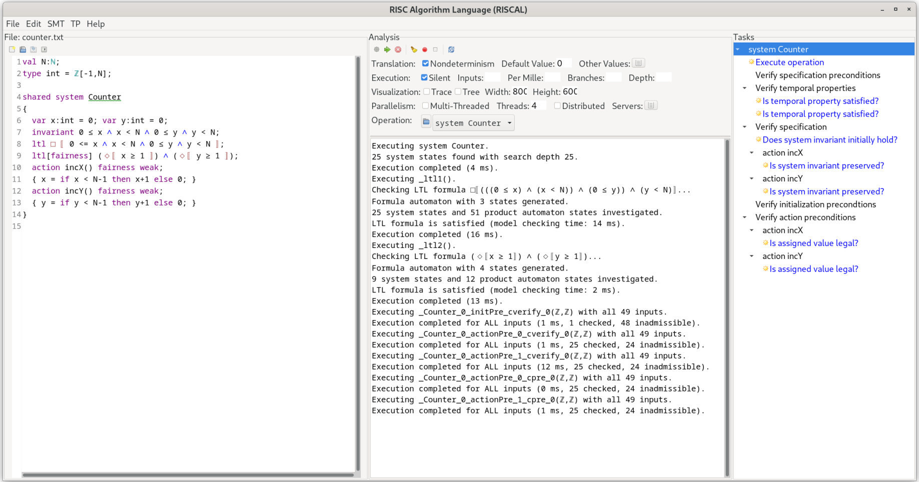
The verification of a safety property by an induction proof.

Transition Systems in RISCAL

```
val N:ℕ;
type int = ℤ[-1,N];

shared system Counter
{
  var x:int = 0; var y:int = 0;
  invariant 0 ≤ x ∧ x < N ∧ 0 ≤ y ∧ y < N;

  ltl □ ⟦ 0 <= x ∧ x < N ∧ 0 ≤ y ∧ y < N ⟧;
  ltl[fairness] (◇⟦ x ≥ 1 ⟧) ∧ (◇⟦ y ≥ 1 ⟧);

  action incX() fairness weak;
  { x = if x < N-1 then x+1 else 0; }
  action incY() fairness weak;
  { y = if y < N-1 then y+1 else 0; }
}
```

Fairness constraints also ensure liveness properties.

Transition Systems in RISCAL



File: counter.txt

```
val N:ℕ;
type int = ℤ[-1,N];

shared system Counter
{
  var x:int = 0; var y:int = 0;
  invariant 0 ≤ x ∧ x < N ∧ 0 ≤ y ∧ y < N;
  ltl □ ⟦ 0 <= x ∧ x < N ∧ 0 ≤ y ∧ y < N ⟧;
  ltl[fairness] (◇⟦ x ≥ 1 ⟧) ∧ (◇⟦ y ≥ 1 ⟧);
  action incX() fairness weak;
  { x = if x < N-1 then x+1 else 0; }
  action incY() fairness weak;
  { y = if y < N-1 then y+1 else 0; }
}
```

Translation: ☑ Nondeterminism Default Value: 0 Other Values:

Execution: ☑ Silent Inputs: Per Mille: Branches: Depth:

Visualization: ☐ Trace ☐ Tree Width: 80C Height: 60C

Parallelism: ☐ Multi-Threaded Threads: 4 ☐ Distributed Servers:

Operation: system Counter

```
Executing system Counter.
25 system states found with search depth 25.
Execution completed (4 ms).
Executing _ltl1().
Checking LTL formula □⟦(((0 ≤ x) ∧ (x < N)) ∧ (0 ≤ y)) ∧ (y < N)⟧...
Formula automaton with 3 states generated.
25 system states and 51 product automaton states investigated.
LTL formula is satisfied (model checking time: 14 ms).
Execution completed (16 ms).
Executing _ltl2().
Checking LTL formula (◇⟦x ≥ 1⟧) ∧ (◇⟦y ≥ 1⟧)...
Formula automaton with 4 states generated.
9 system states and 12 product automaton states investigated.
LTL formula is satisfied (model checking time: 2 ms).
Execution completed (13 ms).
Executing _Counter_0_initPre_cverify_0(ℤ,ℤ) with all 49 inputs.
Execution completed for ALL inputs (1 ms, 1 checked, 48 inadmissible).
Executing _Counter_0_actionPre_0_cverify_0(ℤ,ℤ) with all 49 inputs.
Execution completed for ALL inputs (1 ms, 25 checked, 24 inadmissible).
Executing _Counter_0_actionPre_1_cverify_0(ℤ,ℤ) with all 49 inputs.
Execution completed for ALL inputs (12 ms, 25 checked, 24 inadmissible).
Executing _Counter_0_actionPre_0_cpre_0(ℤ,ℤ) with all 49 inputs.
Execution completed for ALL inputs (0 ms, 25 checked, 24 inadmissible).
Executing _Counter_0_actionPre_1_cpre_0(ℤ,ℤ) with all 49 inputs.
Execution completed for ALL inputs (1 ms, 25 checked, 24 inadmissible).
```

- system Counter
  - Execute operation
  - Verify specification preconditions
  - Verify temporal properties
    - Is temporal property satisfied?
    - Is temporal property satisfied?
  - Verify specification
    - Does system invariant initially hold?
    - action incX
      - Is system invariant preserved?
    - action incY
      - Is system invariant preserved?
  - Verify initialization precondtions
  - Verify action preconditions
    - action incX
      - Is assigned value legal?
    - action incY
      - Is assigned value legal?

Verification by state space exploration and/or induction proofs.