Validating Mathematical Theorems and Algorithms with RISCAL*

Wolfgang Schreiner[0000−0001−9860−1557]

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at
https://www.risc.jku.at

Abstract. RISCAL is a language for describing mathematical algorithms and formally specifying their behavior with respect to user-defined theories in first-order logic. This language is based on a type system that constrains the size of all types by formal parameters; thus a RISCAL specification denotes an infinite class of models of which every instance has finite size. This allows the RISCAL software to fully automatically check in small instances the validity of theorems and the correctness of algorithms. Our goal is to quickly detect errors respectively inadequacies in the formalization by falsification in small model instances before attempting actual correctness proofs for the whole model class.

Keywords: Formal specification · Falsification · Model checking.

1 Introduction

The application of formal methods in computer science and computer mathematics is hampered by the fact that typically most time and effort in proving theorems (such as the verification conditions of a computer program) are wasted in vain: because of errors in the formalizations, proof attempts are often a priori doomed to fail (because the conditions do not hold) or pointless (because the conditions do not express the required intentions).

The RISC Algorithm Language (RISCAL) [3,5] attempts to detect such errors quickly by making algorithms and theorems fully checkable. For this purpose, all (program and logical) variables are restricted to finite types whose sizes are parameterized by formal parameters. For each assignment of concrete values to these parameters, the state space in which algorithms execute and the domain in which first-order formulas are interpreted are finite; this allows to check the correctness of algorithms and the validity of formulas. In RISCAL algorithms may be formulated in a very abstract way including non-deterministic constructions (choices of values satisfying certain conditions). The RISCAL software implements the denotational semantics of programs and formulas in order

* Supported by the Johannes Kepler University, Linz Institute of Technology (LIT), project LOGTECHEDU, and by the OEAD WTZ project SK 14/2018 SemTech.

to perform the checks; in particular, formulas are proved/disproved by "brute-force" evaluation to their truth values. We may thus quickly validate/falsify the correctness of theorems and algorithms in small model instances before turning to their proof-based verification for the full class of models. While [5] gives an extensive overview on (an earlier version) of RISCAL, this paper focuses on its practical use for the purpose of computer mathematics.

Various other modeling languages support checking respectively counterexample generation [2], differing from RISCAL mainly in the application domain of the language and/or the exhaustiveness of the analysis. For instance, the specification language of Alloy is based on a relational logic designed for modeling the states of abstract systems and the executions of such systems; given bounds for state sizes and execution lengths, the Alloy Analyzer finds all executions leading to states satisfying/violating given properties. On the other hand, the counterexample generator Nitpick [1] for the proof assistant Isabelle/HOL supports classical predicate logic but may (due to the presence of unbounded quantifiers) fail to find counterexamples; in an "unsound mode" quantifiers are artificially bounded, but then invalid counterexamples may be reported.

While Alloy and Nitpick are based on SAT-solving techniques, RISCAL's implementation is actually closer to that of the test case generator Smallcheck [4]. This system generates for the parameters of a Haskell function in an exhaustive way argument values up to a certain bound; the results of the corresponding function applications may be checked against properties expressed in first-order logic (encoded as executable higher-order Haskell functions). However, unlike RISCAL, the value generation bound is specified by global constants rather than by the types of parameters and quantified variables such that separate mechanisms have to be used to restrict searches for counterexamples.

RISCAL differs from these approaches in that it combines a rich language that is suitable for specifying and implementing mathematical algorithms (including implicitly specified functions and non-deterministic choices) with a fully checkable semantics, all of which is integrated in a single easy to use specification and validation framework (see Figure 1 for its GUI). RISCAL is intended primarily for educational purposes [6]; it is open source and freely available [3] with extensive documentation and examples.

2 Checking Theorems and Algorithms

We demonstrate the use of RISCAL by a small example adapted from [5].

A Theory The following specification introduces a domain `Formula` of propositional formulas with n variables in conjunctive normal form:

```
val n: ℕ;
type Literal = ℤ[-n,n] with value ≠ 0;
type Clause = Set[Literal] with ∀l∈value. ¬(-l∈value);
type Formula = Set[Clause];
```

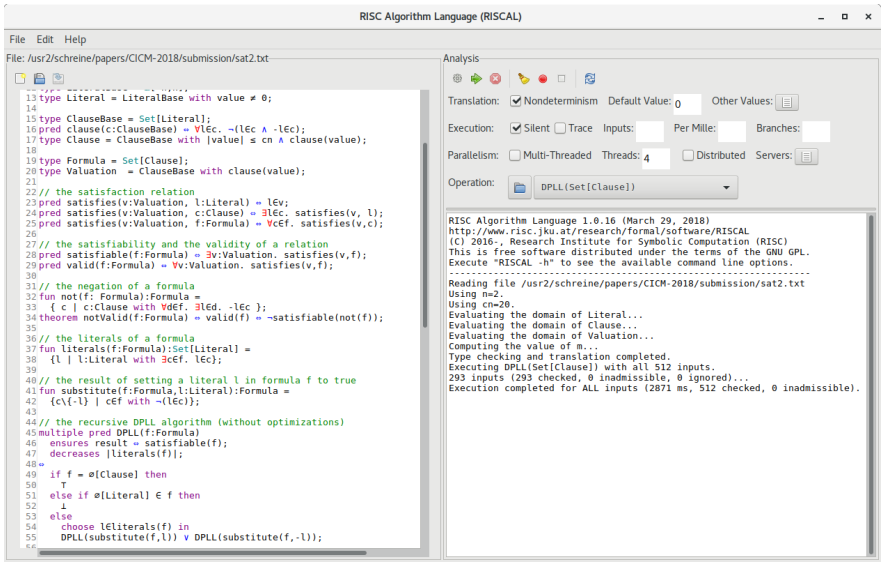

Fig. 1. The RISCAL Graphical User Interface

A formula is represented as a set of clauses where a clause is a set of non-conflicting literals; a literal is a non-zero integer in the interval $[-n, n]$ (in a type definition, `value` denotes the parameter of the predicate constraining the values of the type). A formula valuation has the same representation as a clause: positive literals in the clause receive the value "true" while negative literals receive the value "false". Based on this representation, we can define a predicates `satisfies(v,f)` denoting the relation "valuation v satisfies formula f", `satisfiable(f)` denoting "f is satisfiable", and `valid(f)` denoting "f is valid":

```
type Valuation  = Clause;
pred satisfies(v:Valuation, f:Formula) ⇔ ∀c∈f. ∃l∈c. l∈v;
pred satisfiable(f:Formula) ⇔ ∃v:Valuation. satisfies(v,f);
pred valid(f:Formula) ⇔ ∀v:Valuation. satisfies(v,f);
```

Then we introduce the negation `not(f)` of formula f and formulate a theorem that states that f is valid if and only if its negation is not satisfiable:

```
fun not(f:Formula):Formula = { c | c:Clause with ∀d∈f. ∃l∈d. -l∈c };
theorem notValid(f:Formula) ⇔ valid(f) ⇔ ¬satisfiable(not(f));
```

In RISCAL, a theorem is a predicate that shall be true for all possible arguments. We can quickly check its validity for $n = 2$ by selecting in the RISCAL user interface the operation `notValid` and pressing the "Start Execution" button:

```
Executing notValid(Set[Clause]) with all 512 inputs.
Execution completed for ALL inputs (103 ms, 512 checked, 0 inadmissible).
```

Since the number of formulas grows double-exponentially with the number of literals, we modify the definition of `Clause` (leaving all other definitions unchanged) to consider only clauses with up to cn literals:

```
val cn: ℕ;
type Clause = Set[Literal] with |value| ≤ cn ∧ ∀l∈value. ¬(-l∈value);
```

Using $n = 3$ and $cn = 2$ and selecting the "Multi-Threaded" option in the user interface we can check the correctness of the theorem for a large number of inputs in a still quite manageable amount of time:

```
Executing notValid(Set[Clause]) with all 524288 inputs.
PARALLEL execution with 4 threads (output disabled).
...
Execution completed for ALL inputs (17445 ms, 524288 checked, 0 inadmissible).
```

However, the checking of an invalid "theorem"

```
theorem notValidDual(f:Formula) ⇔ satisfiable(f) ⇔ ¬valid(not(f));
```

produces the expected failure:

```
ERROR in execution of notValidDual({{-3},{-2},{-1}}): evaluation of
  notValidDual
at line 42 in file sat2.txt:
  theorem is not true
```

In order to demonstrate the reason of failures, formulas may be annotated such that the values of subexpressions are printed during evaluation/checking; more elaborate explanation features are planned for the future.

An Algorithm We now consider an algorithm for determining whether a formula f is satisfiable. As a prerequisite, we introduce two operations to determine the set of literals occurring in f and to give a literal l in f the value "true":

```
fun literals(f:Formula):Set[Literal] = { l | l:Literal with ∃c∈f. l∈c };
fun substitute(f:Formula,l:Literal):Formula = { c\{-l} | c∈f with ¬(l∈c) };
```

Now a simple recursive algorithm solving our problem can be defined as follows:

```
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));
```

This algorithm is annotated by the `ensures` clause with the postcondition that the result of the algorithm has to satisfy: it must be "true" if and only if the formula f is satisfiable. Indeed this condition will be checked in all executions and any violations will be reported. Furthermore, the `decreases` clause indicates a measure that is decreased in every recursive invocation but does not become negative: the number of literals in f. Also these constraints will be checked in all executions such that non-termination can be ruled out.

In the recursive case, the algorithm chooses an arbitrary literal l in f and calls itself recursively twice, once with setting l to "true", once with setting l to "false". The RISCAL language has generally a *nondeterministic* semantics: if in the user interface the option "Nondeterminism" is chosen, the algorithm will be executed with *all* possible choices of l. With $n = 2$, we thus get output:

```
Executing DPLL(Set[Clause]) with all 512 inputs.
Execution completed for ALL inputs (884 ms, 512 checked, 0 inadmissible).
```

If this option is not selected, the algorithm will be only executed for one choice, which increases for larger n the speed of the check considerably; e.g., using $n = 3$ and $cn = 2$, we get:

```
Executing DPLL(Set[Clause]) with all 524288 inputs.
PARALLEL execution with 4 threads (output disabled).
...
Execution completed for ALL inputs (196219 ms, 524288 checked, 0 inadmissible).
Not all nondeterministic branches may have been considered.
```

An algorithm may be also formulated in an imperative style as a state-based procedure, again (by the use of a `choose` command) with a non-deterministic semantics. In such procedures, loops may be annotated by invariants and termination measures; also these are checked by all executions of the algorithm. See [5] for more details on the command language for writing state-based procedures.

Validating Specifications As shown above, we may check that the application of a RISCAL procedure p to every argument x that satisfies the procedure's precondition $P(x)$ generates a result y that satisfies the postcondition $Q(x, y)$. However, this check is meaningless if the specification formalized by P and Q does not really express our informal intentions. As a first possibility to validate the adequacy of the formal specification, RISCAL generates from it an implicitly defined function:

```
_pSpec(x) requires P(x) = choose y with Q(x,y);
```

This function can be executed for given x by enumerating all possible values of y (in the result domain) and checking if $Q(x, y)$ holds. Its execution in nondeterministic mode thus produces for every input x constrained by $P(x)$ every value y allowed by $Q(x, y)$; the results of this execution are displayed and may be inspected to confirm the correspondence of the specification to our intentions.

Furthermore, RISCAL automatically generates from P and Q various theorems whose validity may be checked to ensure that the specification satisfies various (usually) expected properties:

- *Is precondition satisfiable?* $(\exists x.\ P(x))$
- *Is precondition not trivial?* $(\exists x.\ \neg P(x))$
- *Is postcondition always satisfiable?* $(\forall x.\ P(x) \Rightarrow \exists y.\ Q(x, y))$
- *Is postcondition always not trivial?* $(\forall x.\ P(x) \Rightarrow \exists y.\ \neg Q(x, y))$
- *Is postcondition at least sometimes not trivial?* $(\exists x.\ P(x) \wedge \exists y.\ \neg Q(x, y))$
- *Is result unique?* $(\forall x, y_1, y_2.\ P(x) \wedge Q(x, y_1) \wedge Q(x, y_2) \Rightarrow y_1 = y_2)$

RISCAL determines the validity of also these formulas by their evaluation.

3 Current and Further Work

For checking a procedure, a loop may be annotated with invariants whose validity is checked before and after every iteration. However, while checking may reveal that an invariant is too strong (it is violated by some loop iteration), it cannot reveal that it is too weak to carry a proof-based verification of the procedure's correctness. Therefore, currently work is under way to generate in RISCAL also *verification conditions* whose validity (with respect to the whole model class) implies the general correctness of the procedure; these conditions can be falsified by checks (in individual instances of the class). If we are not able to falsify these conditions, our confidence in their validity is increased and we may subsequently attempt their proof-based verification (for this we later plan to integrate RISCAL with an external proof assistant).

As for its application in education, so far RISCAL has been used in a fourth year's master course on "Formal Methods in Software Development" in order to make students familiar with the semantics and pragmatics of program specifications before exposing them to proof-based tools; this has indeed helped students to develop adequate program specifications and annotations as a basis for general verification. In the next year, we plan to use RISCAL also in a first semester's introductory course on "Logic" to train students by small exercises in the practical use of first-order logic and in an introductory course on "Formal Modeling" to develop logic-based formal models of problem domains. Our long-term goal is to develop (with the help of students) a collection of educational resources in various areas of computer science and mathematics; we have started to cover selected topics in discrete mathematics, computer algebra, and fundamental algorithms.

References

1. Blanchette, J.C., Nipkow, T.: Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving (ITP 2010). Springer LNCS, vol. 6172, pp. 131–146 (2010). https://doi.org/10.1007/978-3-642-14052-5_11
2. Butler, M., et al. (eds.): Abstract State Machines, Alloy, B, TLA, VDM, and Z, 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings, Springer LNCS, vol. 9675 (2016). https://doi.org/10.1007/978-3-319-33600-8
3. RISCAL: The RISC Algorithm Language (RISCAL) (March 2017), https://www.risc.jku.at/research/formal/software/RISCAL
4. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In: First ACM SIGPLAN Symposium on Haskell. pp. 37–48. Haskell '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1411286.1411292
5. Schreiner, W.: The RISC Algorithm Language (RISCAL) — Tutorial and Reference Manual (Version 1.0). Technical report, RISC, Johannes Kepler University, Linz, Austria (March 2017), download from [3]
6. Schreiner, W., Brunhuemer, A., Fürst, C.: Teaching the Formalization of Mathematical Theories and Algorithms via the Automatic Checking of Finite Models. In: Post-Proceedings ThEdu'17, Theorem proving components for Educational software. EPTCS, vol. 267, pp. 120–139 (2018). https://doi.org/10.4204/EPTCS.267.8