Mathematical Model Checking Based on Semantics and SMT

Schreiner, Wolfgang; Reichl; Franz-Xaver

Abstract: ***We report on the usage and implementation of RISCAL, a model checker for mathematical theories and algorithms based on a variant of first-order logic with finite models; this allows to automatically decide the validity of all formulas and to verify the correctness of all algorithms specified by such formulas. We describe the semantics-based implementation of the checker as well as a recently developed alternative based on SMT solving, and experimentally compare their performance. Furthermore, we report on our experience with RISCAL for enhancing education in computer science and mathematics, in particular in academic courses on logic, formal methods, and formal modeling. By the use of this software, students are encouraged to actively engage with the course material by solving concrete problems where the correctness of a solution is automatically checked; if a solution is not correct or the student gets stuck, the software provides additional insight and hints that aid the student towards the desired result.***

Index Terms: ***model checking, logic, semantics, formal verification, reasoning about programs, computer science education***

1. Introduction

SOFTWARE based on formal logic plays an ever-increasing role in areas where a mathematically precise understanding of a subject domain and sound rules for reasoning about the properties of this domain are essential. A prime example is the formal modeling, specification, and verification of computer programs and computing systems, but there are many other applications in areas such as knowledge-based systems, computer mathematics, or the semantic web.

As for reasoning in these domains, there are two main approaches, *proving* and *model-checking*. The main advantage of proof-based systems [6], [9] is their generality: they can operate with rich formal systems such as first-order logic and reason about domains of infinite size. Their disadvantage, however, is that such rich logics are generally undecidable such that the failure to construct a proof for a conjecture does not indicate its invalidity. This is a major problem in areas such as program verification where these conjectures are verification conditions derived not only from the specifications of the program but also from extra (in practice human-provided) information such as loop invariants; if these invariants are too strong or too weak, the verification conditions are not valid, even if the program satisfies its specification.

The main advantage of model checkers [2] is their automatism: they decide without user assistance the validity of conjectures; furthermore, if a conjecture is invalid, they produce a counterexample that demonstrates its invalidity. Their disadvantage is that they have to be based on decidable calculi, which usually entails finite domains of interpretation or weaker formal systems than first-order logic. For instance, bounded model checkers consider only finite (prefixes of) program runs and typically only check for specific properties such as preconditions of built-in operations.

In this paper, we present an approach that combines the generality of a rich logic with the automation of model checking. This approach is implemented in *RISCAL* [11], [12], [16], a software system for the formalization of mathematical theories and the specification and verification of algorithms operating in such theories. The software also provides various means to aid the understanding of results, e.g., by producing counterexamples or by the graphical visualization of evaluation trees [15]. The checking mechanism is based on an executable version of the denotational semantics of the specification language; recently we have developed an alternative approach based on the translation of this language to the language SMT-LIB supported by various satisfiability modulo theories solvers [10].

Various other modeling languages also support checking but differ from RISCAL in the application domain of the language or the exhaustiveness of the checks. Alloy [4] is based on a relational logic designed for modeling abstract systems but little suitable for mathematical theories. TLA$^+$ [5] embeds first-order logic but is untyped and also supports models of infinite size; thus, the TLA$^+$ toolkit does not really represent a reliable decision procedure. In contrast, RISCAL is based on full first-order logic and also allows, e.g., implicitly defined

Manuscript received June 8, 2020. This work was supported by the Johannes Kepler University Linz, Linz Institute of Technology, Project LOGTECHEDU "Logic Technology for Computer Science Education" and by the OEAD WTZ project SK 14/2018 SemTech.

W. Schreiner (contact person) is an associate professor at the Research Institute for Symbolic Computation of the Johannes Kepler University Linz, Austria (e-mail: Wolfgang.Schreiner@risc.jku.at).

F.-X. Reichl is a student of computer mathematics at the Johannes Kepler University, Linz, Austria (e-mail: franz.x.reichl@gmail.com).

functions and nondeterministic computations; still by its restriction to models of (parametrizable) finite size, the validity of formulas and the correctness of algorithms is fully decidable [13].

A main application of RISCAL is in educational scenarios where questions and problem statements have a precise and machine-understandable meaning and where the correctness of answers and problem solutions can be automatically checked. The use of this software gives students the possibility to self-check the correctness of their solutions and to use the feedback of the software to correct their errors. The goal is a style of "self-directed learning" where students do not just passively consume educational material but actively interact with it by producing problem solutions with the help of software. Ultimately such software might also be integrated into software for the automatic grading of assignments [20].

In this paper, we describe our actual experience with the use of RISCAL in academic courses on formal specification and verification, formal modeling, and logic; these courses have addressed various types of audiences, computer scientists as well as mathematicians, from absolute beginners to master students in the later phases of their education. RISCAL has been partially developed in the context of two projects, "Logic Technologies for Computer Science Education" (LogTechEdu) at Johannes Kepler University (JKU) Linz [7], and "SemTech" at JKU Linz and the Technical University (TU) of Košice [17], that investigate and further develop such tools for their use in computer science education. An example for other software developed in these projects is the toolset Jane [18], [19] that illustrates graphically the formal semantics of a simple programming language and that has been applied in various courses.

The remainder of this paper is structured as follows: In Section 2 we sketch the use of RISCAL on small examples. Section 3 discusses the semantics-based implementation of the RISCAL checker, the alternative method based on SMT solving, and their experimental comparison. In Section 4 we describe our experience with the use of RISCAL in education. Section 5 presents our conclusions and outlines further work.

This paper is a revised version of [14] extended by previously unpublished material (Section 3).

2. The RISCAL Software

RISCAL (RISC Algorithm Language) is a language and associated software system for formulating theories in first-order logic, describing algorithms in a high-level language, and specifying the behavior of these algorithms by formal contracts. The language is based on a type system where all types have finite sizes (specified by the user); this allows to fully automatically decide formulas and to verify the correctness of algorithms for all possible inputs. To this end, the system translates every syntactic phrase into an executable form of its denotational semantics; the RISCAL model checker evaluates this semantics to determine the results of algorithms and the truth values of formulas such as the postconditions of algorithms. Figure 1 displays the user interface of the software with an editor panel on the left and control elements and an output terminal on the right.

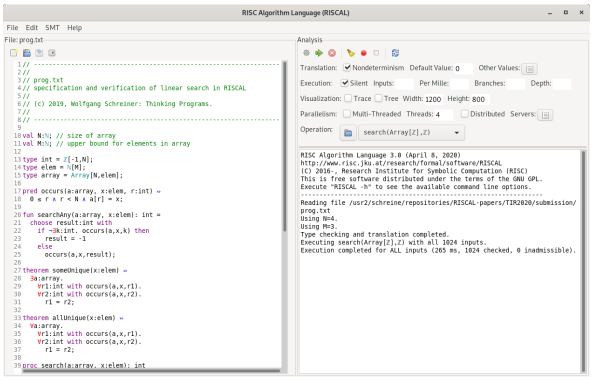

Fig. 1. The RISCAL Graphical User Interface

As an example for the use of RISCAL, take the following constant and type declarations:

```
val N:ℕ; val M:ℕ;
type int = ℤ[-1,N]; type elem = ℕ[M];
type array = Array[N,elem];
```

These introduce a type $array$ of arrays that have some length N and elements of type $elem$; each such element is a natural number up to some maximum M. Likewise an auxiliary type int of integers from -1 to N is defined. Furthermore, we introduce by the definition

```
pred occurs(a:array, x:elem, r:int) ⇔
  0 ≤ r ∧ r < N ∧ a[r] = x;
```

a predicate $occurs$ that is true if array a holds element x at position r. From this, the declaration

```
fun searchAny(a:array, x:elem): int =
  choose result:int with
    if ¬∃k:int. occurs(a,x,k) then
      result = -1
    else
      occurs(a,x,result);
```

introduces an implicitly defined function $searchAny$ that returns an arbitrary position at which x occurs in a, respectively the value -1, if x does not occur in a. RISCAL can execute this definition, e.g., for $N = 4$ and $M = 3$; if we select for this execution the "nondeterministic" mode, for every input *all* possible outputs are determined:

```
Executing searchAny(Array[ℤ],ℤ) with
  all 1024 inputs.
Branch 0:0 of nondeterministic function
  searchAny([0,0,0,0],0):
Result (0 ms): 0
...
Branch 4:1023 of nondeterministic ...
  searchAny([3,3,3,3],3):
No more results (15 ms).
```

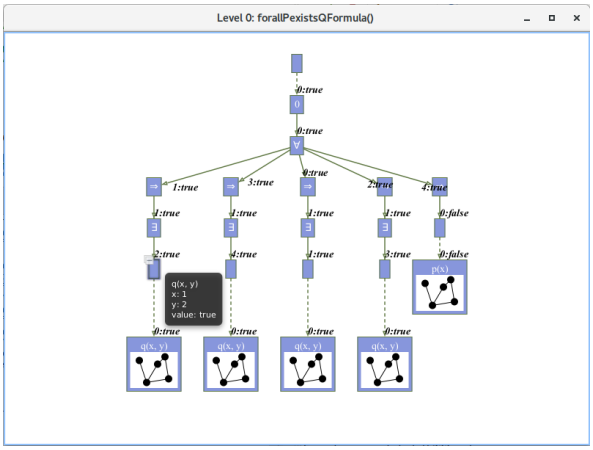

Fig. 2. A Pruned Formula Evaluation Tree in RISCAL

```
Execution completed for ALL inputs
  (6741 ms, 1024 checked, ...).
```

This is possible, because in nondeterministic mode, the semantics of functions, predicates, and procedures does not denote a single value but a finite set of values which is implemented by a lazily evaluated stream (see Section 3).

Based on the predicate $occurs$, we may define and check the (apparently valid) theorem that for every element x there is *some* array that holds x at a unique position:

```
theorem someUnique(x:elem) ⇔ ∃a:array.
  ∀r1:int with occurs(a,x,r1).
  ∀r2:int with occurs(a,x,r2). r1 = r2;
Executing someUnique(ℤ) with all 4 ...
Execution completed for ALL inputs ...
```

However, we may also define and check the (invalid) theorem that above is true for every array:

```
theorem allUnique(x:elem) ⇔ ∀a:array.
  ∀r1:int with occurs(a,x,r1).
  ∀r2:int with occurs(a,x,r2). r1 = r2;
Executing allUnique(ℤ) with all 4 ...
ERROR in execution of allUnique(0):
evaluation of
  allUnique
at line 33 in file prog.txt:
  theorem is not true
ERROR encountered in execution.
```

We may then ask for a counterexample:

```
Executing allUnique_refute().
This sequence of variable assignments
  leads to a counterexample ...:
x=0
a=[0,0,0,0]
r1=0
r2=1
Execution completed (5 ms).
```

The truth values of formulas can be also visualized in RISCAL by the help of “pruned evaluation trees” (see Figure 2 for an example of such a tree) that depict those paths in the evaluation of quantified formulas that determine the overall outcome.

After these preliminaries, we define a deterministic procedure that returns the *smallest* position r of element x in array a; if x does not occur in a, the procedure returns -1:

```
proc search(a:array, x:elem): int {
  var i:int = 0; var r:int = -1;
  while i < N ∧ r = -1 do {
    if a[i] = x
      then r := i;
      else i := i+1;
  }
  return r;
}
```

To verify its correctness, we annotate the procedure with the following postcondition:

```
ensures if ¬∃k:int. occurs(a,x,k) then
  result = -1
else occurs(a,x,result) ∧
  ∀k:int with occurs(a,x,k). result≤k;
```

We may then check the correctness of the algorithm for all possible inputs:

```
Executing search(...) with all 1024...
Execution completed for ALL inputs ...
```

However, if we replace the test $a[i] = x$ erroneously by $a[i] \neq x$, we get the following error:

```
ERROR in execution of search(...):
evaluation of ensures ... at line ...:
  postcondition is violated by result
    -1  for application search(...)
```

Furthermore, we may annotate the above loop by an invariant and termination measure:

```
invariant 0 ≤ i ∧ i ≤ N;
invariant
  ∀k:int. 0 ≤ k ∧ k < i ⇒ a[k] ≠ x;
invariant
  r = -1 ∨ (r = i ∧ i< N ∧ a[r] = x);
decreases if r = -1 then N-i else 0;
```

Every execution of the procedure checks the validity of these annotations; this in particular ensures that the given invariant is not too strong. However, we may also let the system generate from this invariant verification conditions whose validity implies the correctness of the program. These conditions are theorems that can be automatically checked in RISCAL by a single mouse-click (see Figure 3), which ensures that the specified invariants are strong enough (they are “inductive”). Thus, we may subsequently use some other environment to verify the correctness of the algorithm by formal proof for *arbitrary* values of M and N.

The big advantage of RISCAL is that it allows to formulate rich formal/mathematical/logical contents (theories and algorithms) in an expressive language (first order logic including expressions that do not necessarily have unique values) and still have their adequacy fully automatically checked over small domains. In this way, errors in the formulations can be easily caught and thus the formalism be quickly validated; this is not so simply possible with proof-based approaches where failed proof attempts more often than not indicate the inadequacy of proof strategies rather than the invalidity of proof goals. Only when we are after such a validation reasonably convinced about the

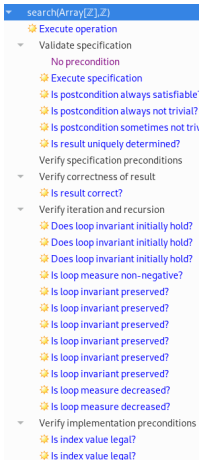

Fig. 3. The Validity of Verification Conditions in RISCAL

correctness of the formulations, we will turn to proof-based verification over general domains.

3. Checking and SMT Solving in RISCAL

The basic mechanism of RISCAL for checking the correctness of algorithms and theorems relies on the evaluation of their denotational semantics; however in [10] an alternative mechanism is presented which is based on the translation of RISCAL formulas to the language SMT-LIB supported by various satisfiability modulo theories (SMT) solvers. In this section, we sketch both approaches and compare their performance by a number of benchmarks.

3.1. Model Checking

RISCAL implements an executable form of the denotational semantics of every kind of phrase of its language. If a phrase is deterministic, i.e., its evaluation yields a uniquely defined result, its semantics is typically an element of the domain

$$Single[T] := Context \to T$$

i.e., it maps a context (which determines the values of the free variables of the phrase) to a value of the semantic domain T. However, a phrase may also be nondeterministic, i.e., its evaluation may yield multiple results. Then its semantics is typically an element of the domain

$$Multiple[T] := Context \to Seq[T]$$

where $Seq[T]$ is a collection of T-values which we will for the moment consider as a set (more details will be given below).

For instance, in the case of imperative commands we have $T := Context$, i.e., the result is another context (which denotes the values of the variables after the execution of the command); in the case of logic formulas we have $T := Bool$, i.e., the result is a truth value; in the case of terms we have $T := Value$, i.e., the result is a value of the object domain of the logic.

The denotation of a phrase of syntactic domain S is defined by a mapping $[\![.]\!] : S \to Single[T] + Multiple[T]$. For instance, assuming two deterministic commands C_1 and C_2, the deterministic semantics $[\![C_1 ; C_2]\!] \in Single[Context]$ of their composition can be defined as follows:

$$[\![C_1 ; C_2]\!] := \lambda c.\ \text{let}\ c_1 = [\![C_1]\!](c)\ \text{in}\ [\![C_2]\!](c_1)$$

However, if C_1 and C_2 are nondeterministic, then we have the nondeterministic semantics $[\![C_1 ; C_2]\!] \in Multiple[Context]$ defined as follows:

$$[\![C_1 ; C_2]\!] := \lambda c.\ \text{let}\ cs_1 = [\![C_1]\!](c)\ \text{in} \bigcup_{c_1 \in cs_1} [\![C_2]\!](c_1)$$

RISCAL implements these mathematical notions by corresponding constructions in Java. For instance, we have data types like

```
public interface Single<T> extends
  Function<Context,T> { }
public interface Multiple<T> extends
  Function<Context,Seq<T>> { }
```

The domain $Seq[T]$ is implemented as a class with an object method `get()` that returns either `null` (the end of the sequence) or a pair of a sequence element and another object of this class. Thus $Seq[T]$ actually denotes the domain of *lazily* evaluated sequences of T-values as the basis of the nondeterministic execution semantics. The implementation of the (deterministic/nondeterministic) semantics of phrases such as command sequences is based on functions like

```
static Single<Context>
seqCommand(Single<Context> C1,
           Single<Context> C2)
{ return (Context c) ->
    { Context c1 = C1.apply(c);
      return C2.apply(c1); } }
static Multiple<Context>
seqCommand(Multiple<Context> C1,
           Multiple<Context> C2)
{ return (Context c) ->
   { Seq<Context> cs1 = C1.apply(c);
     return cs1.mapJoin(C2); } }
```

The whole implementation heavily depends on the lambda expressions introduced in Java 8; in fact, the RISCAL implementation of the denotational semantics resembles very much a (higher-order) functional program.

RISCAL first builds from the concrete syntax of a phrase an abstract syntax tree; then it applies a type checker to annotate this tree with symbolic information determined by static analysis; then it translates the annotated tree to its denotational semantics; finally, it executes this semantics of the phrase. Since modern JIT compilers heavily optimize at runtime the executions of the functional objects resulting from this translation, we get an evaluation mechanism for the various phrases, which is much more efficient than classical “interpretation”; in fact, the translation mechanism

achieves many of the advantages we get by "compilation" to a low-level machine language.

If the phrase is a formula, we thus have (since every RISCAL type is finite) a decision procedure for the validity of the formula. If the phrase is a program, we have an execution mechanism for the program; if the program is annotated with meta-information such as preconditions, postconditions, loop invariants, and termination measures, we have a model checker for the correctness of the program. Since RISCAL also implements a verification condition generator where the derived verification conditions are again RISCAL formulas, we thus also have a (limited form of) a program verification environment. From RISCAL Version 3 on, the language also supports the concept of concurrent systems whose semantics are transition systems; the evaluation of these systems yields a model checker for the verification of the invariance of safety conditions.

3.2. SMT Solving

The application of SMT solvers in RISCAL, presented in [10], relies on the translation of a theorem (and of the specification on which the theorem depends) into an SMT-LIB [1] script. For this purpose, we use the SMT-LIB logic of Quantifier Free Formulas with Fixed Size Bit Vectors and Uninterpreted Functions (QF_UFBV). This idea of a translation is related to [8] where a translation of TLA$^+$ enabled the application of SMT solvers.

The translation ensures that the original RISCAL theorem is valid if and only if the generated SMT-LIB formula is unsatisfiable. This can be decided by a variety of SMT solvers supporting the QF_UFBV logic, in particular Boolector, CVC4, Yices 2, and Z3. As already discussed above, RISCAL provides the functionality to generate verification conditions for algorithms. Since these conditions are just theorems in RISCAL, they can be also decided by the translation to SMT-LIB.

In the remaining part of this section, we will discuss selected aspects of this translation which proceeds in two steps:

1) We translate the input into a form that makes the later encoding substantially easier and negate the given theorem.
2) We encode the preprocessed RISCAL specification into SMT-LIB such that satisfiability of the negated theorem is preserved.

The translation step, e.g., renames overloaded functions such that every function gets a unique name. We also eliminate several kinds of expressions by rewriting them into logically equivalent forms; in particular we replace "choose expressions" (nondeterministic choices) by applications of newly introduced "choice functions", whose interpretations are restricted by appropriate axioms. For example, the theorem:

```
theorem T(x:ℕ[N]) ⇔ x ≥
  choose y:ℕ[N] with x=2·y ∨ x=2·y+1;
```

is preprocessed to:

```
fun chooseFun(x:ℤ[0,N]):ℤ[0,N]
axiom chooseAx ⇔ ∀x:ℤ[0,N].
  (∃y:ℤ[0,N]. (((x = (2·y)) ∨
  (x=((2·y)+1))) ∧ (chooseFun(x)=y)));
theorem T ⇔ ∃x:ℕ[N].(x<chooseFun(x));
```

The encoding step performs two tasks:

1) We eliminate all universal and existential quantifiers because the target of our translation (QF_UFBV) is a quantifier-free logic. To eliminate existential quantifiers, we mainly apply the technique of "skolemization", i.e., we replace existentially quantified variables by applications of uninterpreted "Skolem functions". However, since the result is only an equi-satisfiable (not a logically equivalent) formula, this transformation is only possible within the negated theorem whose satisfiability is to be decided.
   As a more general technique, we replace universally and existentially quantified formulas by expanding them into conjunctions respectively disjunctions of instantiations of the body formula for all values of the type of the quantified variable.
2) We encode the types and operations of RISCAL in the theory of bit vectors. While this translation is generally based on commonly known representations, major adaptations were necessary, e.g., to deal appropriately with integer domains of varying sizes, and to devise effectively computable mappings between the RISCAL types and the bit vector domains.

In the following we illustrate the translation on the basis of the following RISCAL specification:

```
type nat=ℕ[100];
type set=Set[nat];
fun diff(A:set,B:set):set=(A\B)∪(B\A);
theorem assoc(a:set,b:set,c:set)⇔
  diff(diff(a,b),c)=
  diff(a,diff(b,c));
```

Here we introduce a type `set` of sets that have elements of type `nat`. Furthermore, we introduce with `diff` a function that represents the symmetric difference of two sets. Finally, we introduce the theorem `assoc` that states that the symmetric difference is associative (obviously this is a valid theorem). We translate this RISCAL specification to the following SMT-LIB script:

```
(set-logic QF_UFBV)
(define-sort nat ()(_ BitVec 7))
(define-sort set ()(_ BitVec 101))
(define-fun diff ((A (_ BitVec 101))
  (B (_ BitVec 101)))(_ BitVec 101)
  (bvor (bvand A (bvnot B))
  (bvand B (bvnot A))))
(declare-fun f ()(_ BitVec 101))
(declare-fun f_1 ()(_ BitVec 101))
(declare-fun f_2 ()(_ BitVec 101))
```

TABLE 1
SMT PERFORMANCE

Model	Theorem	Values	RISCAL	Yices
catlogic	Imp1	N=2, M=1	71748	92
sets	associativeUnionT	N=8	235413	2
gcd2	_gcdf_8_PostUnique	N=100	Timeout	55
bubble	_swap_0_CorrOp0	N=6, M=6	253559	6
sat3	_DPLL2_14_CorrOp0	n=2, cn=2	Timeout	10
matrices	symAdd	N=4	Timeout	4
graphs	_handshaking Theorem_8_PreSat	N=3	0	9682
sat3	_DPLL2_14_PostNot TrivialSome	n=3, cn=2	0	920
bubble	_bubbleSort3_2 _CorrOp0	N=4, M=4	10399	25695
search	_bsearchp_1_CorrOp3	N=4, M=4	205	4635

```
(assert (let ((a f))(let ((b f_1))
  (let ((c f_2))(distinct
  (diff (diff a b) c)
  (diff a (diff b c)))))))
(check-sat)(exit)
```

Here we introduce sorts `nat` and `set` that represent the same named RISCAL types as well as a function `diff` that represents the corresponding RISCAL function. In the definition of this function we can see that we represent the union of sets by bitwise disjunction and the set difference by means of bitwise negation and conjunction (as $A \setminus B = A \cap B^c$). Finally, we represent the negation of the theorem `assoc`. For this purpose, we first introduce three nullary functions for the skolemization of the quantified variables a, b, and c (as we negate the theorem, we have existentially quantified variables). In the `assert` statement we then state the negated theorem.

If we apply an SMT solver to this SMT-LIB script, it reports that the formula is unsatisfiable, which shows that the theorem `assoc` is valid. Indeed, the SMT solver Yices 2 reports a result almost instantaneously, whereas the basic checking mechanism of RISCAL does not produce an answer in a reasonable amount of time.

3.3. Experimental Comparison

In [10] we systematically compared the performance of the checking mechanisms presented in the previous two sections. For this purpose, we selected a variety of RISCAL specifications respectively theorems. On the one hand, we checked these theorems with the semantics-based mechanism of RISCAL. On the other hand, we applied the SMT-LIB translation and invoked the SMT solvers Boolector, CVC4, Yices 2, and Z3, on the generated scripts. Finally, we compared the obtained results. The conducted tests showed that in general Yices 2 [3] delivered the best results in terms of performance. Thus, here we only present the results of the tests performed with Yices 2.

In Table 1 we give a (small) selection of these results achieved with checking/deciding a variety of theorems that are all valid and that cover a variety of types, e.g., integers, maps/arrays, and sets. The entries of this table are to be read in the following way. Column 1 gives the name of the tested RISCAL specification. Column 2 gives the tested theorem. Column 3 gives the used model parameters. Column 4 gives the time needed by the semantics-based checking mechanism of RISCAL. Column 5 gives the time needed by the translation and the SMT solver Yices 2 to decide the given theorem. All figures represent wall clock times in milliseconds, measured on an Intel Xeon Gold 6128 processor with 1.5 TB of memory. A "timeout" is reported if the checking time exceeded a threshold of twenty minutes. The used RISCAL specifications and the generated SMT-LIB scripts are publicly available at https://www.risc.jku.at/research/formal/software/RISCAL/papers/thesis-Reichl.tgz.

The first six examples of Table 1 illustrate the speedups that can be achieved through the usage of SMT solvers. We want to point out that the tests presented in [10] indicate that such speedups cannot only be achieved for few selected cases, but for a broad variety of RISCAL specifications and theorems. Indeed, in approximately 75% of the tests, the translation used together with the application of Yices 2 was substantially faster than the semantics-based mechanism of RISCAL.

However, while often significant speedups can be achieved by the usage of SMT solvers, there are also cases where the semantics-based checking mechanism delivers superior results. We detected certain patterns in RISCAL specifications that seem to have a negative influence on the performance of the SMT based checking mechanism. The last four examples of Table 1 illustrate two such patterns that may make the usage of the translation disadvantageous.

Examples 7 and 8 illustrate that theorems that contain existentially quantified formulas may be checked significantly faster by the semantics-based method. There are, in particular, two reasons for this. On the one hand, RISCAL can often find a witness for an existential quantifier very fast. Thus, it can often report the validity of theorem very soon — like in the mentioned examples. On the other hand, the translation has to negate the theorems. Thus, existential quantifiers are transformed to universal quantifiers. But universal quantifiers have to be expanded, which can result in large formulas that seem to be disadvantageous for all the tested SMT solvers.

Examples 9 and 10 illustrate that RISCAL specifications with "choose expressions" may be disadvantageous for the application of SMT solvers. To justify this assertion, we have to consider that such expressions are translated to applications of uninterpreted functions. This, for example, means that `choose x:T`, where T is some type, is represented by a function, whose image is the set of

bit vectors that represent the type T. Often not all the bit vectors from the image of such a function correspond to a value that can actually be attained by the original choice. Therefore, it is necessary to use certain assertions that guarantee that the function only attains values that correspond to values that can be given by the choice. These assertions often seem to have a negative influence on the performance of the SMT solvers.

To sum up, the comparison of the two decision mechanisms shows that the SMT based mechanism has the potential to significantly speedup the decisions of the validity of formulas. Still, it cannot fully replace the semantics-based checking mechanism of RISCAL, as certain specification patterns favor the usage of this mechanism.

4. Logic and Semantic Software for Education

The intent of our projects LogTechEdu and SemTech is to further advance education in computer science and related topics: by utilizing the power of modern software based on formal logic and semantics, students shall engage with the material they encounter by actively producing problem solutions rather than just passively consuming them from the lecturer. For this purpose, we have experimented with various pieces of related software respectively further developed such software. For instance, David Cerna has in the frame of LogTechEdu at JKU Linz recently developed an Android app "AXolotl" for the touch-based training of first-order reasoning with term matching respectively substitution; likewise, William Steingartner and Valerie Novitzká have in the frame of SemTech at TU Košice developed the toolset "Jane" for the semantics-based execution and visualization of a simple procedural language that has been successfully employed in various courses on programming language semantics.

In this paper we will discuss in more detail those activities that the first author has been directly connected to, mostly related to the RISCAL software that was described in the previous sections.

a) Formal Specification and Verification: Since 2005, the first author has given at JKU Linz a yearly 4.5 ECTS course on "Formal Methods in Software Development"; the goal is to educate master students in computer science and computer mathematics in the formal specification and verification of computer programs with the help of various freely available software environments. Since 2009 we have also used our own "RISC ProofNavigator" and since 2011 our "RISC ProgramExplorer" for the purpose of verifying programs by deriving and proving verification conditions. In this course groups are of moderate size (about 25 participants) and have already considerable technical background (their formal background, however, is varying).

However, by relying on proof-based verification tools, the adequacy of formal specifications and annotations (loop invariants) could be only judged by proving the validity of the generated verification conditions. If such proofs did not immediately succeed (after applying some standard interactions with the respective proof assistants), many students were not really able to deduce from the failed proof attempt whether this was due to an inadequate proof strategy or due to deficiencies in the specifications/annotations; moreover, sometimes verification attempts trivially succeeded because preconditions were unsatisfiable or postconditions were generally valid.

In 2017, we introduced RISCAL into the course, replacing some of the initial use of the RISC ProofNavigator/ProgramExplorer. In a first exercise students had to validate specifications with the help of various techniques integrated into RISCAL, such as executing implicitly defined functions that were automatically generated from procedure contracts or checking whether pre- and postconditions satisfied various consistency criteria. In later exercises, students had to annotate procedures with invariants and termination measures, and check the verification conditions generated by RISCAL. If conditions were not valid, the RISCAL trace/visualization features could be applied to determine the sources of the errors.

Final anonymous evaluations of the course software performed in 2018 and 2019 indicated a very high satisfaction of students with RISCAL concerning its ease of use and learning success; the ratings were significantly better than for the proof-based RISC ProofNavigator/ProgramExplorer and summarily higher than for the six other toolsets used in the course (the extended static checker ESC/Java2 being the second most popular one). However, there was no objectively visible effect on exercise grades, which remained mostly in the 80–95% (good to very good) range.

Also outside of JKU Linz, at the Czech Technical University in Prague, Stefan Ratschan used in 2019 RISCAL in a 4.5 ECTS course on "Formal Methods and Specification" for 80 students. The software was applied with very good success; apart from some feedback for improving usability and requests for additional features, no problems were reported; in 2020, RISCAL has been employed in this course again.

Apart from courses, two bachelor students of technical mathematics used RISCAL to elaborate in their bachelor theses the formal specification and verification of algorithms from discrete mathematics (mostly relating to set and graph theory) respectively for searching and sorting of sequences in various representations (including the major asymptotically fast algorithms). Especially in the latter case, it was astonishing to see that the stu-

dent, without prior expertise in formal verification, was able to come up with sufficiently strong loop invariants to let the verification succeed. These are (anecdotal but nevertheless) strong indications that students that already have a certain background are indeed able to develop formally adequate theories and specifications.

b) Formal Modeling: In 2019, at JKU a new 3 ECTS course "Formal Modeling" for bachelor students of technical mathematics was introduced, as well as an accompanying proseminar. The course was given in three modules by three lecturers; in the module "Logic Models of Problems and Computations" of that course (in which about 15 students participated), we applied RISCAL to model classical "computational" problems but also "dynamic" search and scheduling problems, often disguised in the form of "puzzles" such as, e.g., the well-known "goat, wolf, and cabbage" river crossing puzzle.

In contrast to the computational problems specified by a pair of pre- and postconditions, the dynamic problems were modeled in RISCAL (which at that time did not yet have direct support for such systems) by nondeterministic algorithms of the following structure:

```
proc system(s0:State):
    Tuple[ℕ[N],Array[N,State]]
  requires init(s0);
{
  var s:State = s0;
  var i:ℕ[N] = 0;
  var t:Array[N,State] =
    Array[N,Action](s);
  while ¬goal(s) ∧ i < N do
  {
    choose s1:State with next(s,s1);
    s = s1; i = i+1; t[i] = s;
  }
  return ⟨i,t⟩;
}
```

The predicate $init$ constrains the initial state of the system; the predicate $goal$ describes the desired goal state. Starting with the initial state s_0 as the current state s, the program nondeterministically chooses a successor state s_1 that is related to s by the relation $next$. The computation terminates with the trace t of the states traversed when a desired goal state has been found or a bound for the number i of steps has been reached (to reduce the search space, typically the computation of the successor state is split into the nondeterministic choice of an "action" a and the subsequent deterministic computation of s_1 from s and a).

Students were handed out specification templates with all the necessary declarations; their task was to formalize the $goal$ predicate and the $next$ relation. By running the procedure $system$ in nondeterministic mode, the adequacy of the definitions could be evaluated.

Students apparently liked the "puzzle-like" nature of these problems; also, because of the possibility of self-checking, the submitted solutions were indeed mostly correct. In the proseminar (attended by five students), two students modeled self-selected problems, in one case the card game "Uno", in the other case the problem of the minimization of finite state automata.

All in all, we encountered the use of RISCAL in this novel way a success; it also demonstrated nicely how by the utilization of nondeterministic choices models of "computational systems" can be constructed, for which RISCAL was originally not designed. A crucial point, however, here was the appropriate modeling of the system to manage the exponential explosion of the search space of nondeterministic choices.

c) Logic: The courses presented so far mainly dealt with moderately sized groups of students with some prior technical and formal knowledge. However, since 2013 we are at JKU also (together with three other lecturers) engaged in a 4.5 ETCS course "Logic" for first semester bachelor students of computer science; this course is attended by 250–350 students most of which have just passed their high school exam, not all with a technical focus. The course is internally organized in three modules "SAT" (propositional logic and satisfiability solving), "FO" (first order predicate logic) and "SMT" (satisfiability modulo theories); the FO module takes half of the course.

Over the years we have also integrated more and more the use of logic-based software tools into the course, a SAT solver (Limboole), an interactive proving assistant (RISC ProofNavigator), an automated prover (Theorema), and SMT solvers (Boolector, Z3). However, prior to 2018, we confined the use of these tools to three optional "laboratory" assignments that students could perform on interest and/or as a substitute for three instances of weekly tests. The main reason was that we could not spend adequate time with the explanation and support of this software and thus did not want to make the use of the software mandatory; consequently, however, only a minority of 5–10% of the students used the software, mainly as a substitute for failed tests. To bring the software more into the "main stream" of the course, we introduced in 2018 weekly "bonus" assignments by which students could earn up to 20% of the grade points for the forthcoming tests. These assignments were of comparatively low complexity; they were mainly intended to raise more interest in the software and thus in the practical aspects of the course.

Also starting with the mentioned year, RISCAL replaced the RISC ProofNavigator in the second module FO, specifically in one laboratory assignment and three of the bonus assignments. Questions were handed out in the form of RISCAL skeleton files in which students had to fill in some

missing parts; RISCAL itself was provided in the form of a virtual machine (to be downloaded and executed in the free VirtualBox environment) and on a remote server (to be used via an X2go client). In a “syntax” assignment students had to “parenthesize” formulas to make their structure unique (RISCAL checked their equivalence to the original unparenthesized formulas) and to translate informal statements into formal ones (RISCAL checked the equivalence to another formalization). In a “semantics” assignment, students had to determine satisfying assignments of first-order formulas (RISCAL checked the correctness of the answers) and to transform formulas into logically equivalent forms with certain syntactic constraints (RISCAL checked here the equivalence). In a “pragmatics” assignment, students had to translate given informal problem specifications (pre- and post-conditions) into logic formulas, which partially involved the definition of auxiliary functions and predicates; here RISCAL was used to validate the results by, e.g., checking the input/output behavior of a function implicitly defined by this condition.

At the end of the course, an anonymous evaluation gave the following results: about 40% of the students performed at least one RISCAL exercise, about the same number reported the use of the software as helpful. These numbers clearly trail the SAT solver Limboole used in the SAT module (about 60% used this one and reported it as helpful) but are also much ahead of the other tools in FO and SMT (used by about 25%). Of those who submitted bonus assignments, most indeed earned the full amount of potential grade points. As for the more general questions on why students used the software, twice as many reported as the main reason to earn the bonus points rather than intrinsic interest. Still, most positive impact on interest was reported to the software, while most impact on understanding was attributed to the exercises (three times more than to software). While the overall level of grades did not significantly differ from the previous years, we found a strong correlation between performance on bonus assignments and performance in classroom assignments; indeed, most students that failed the course did not perform the bonus assignments.

Thus, in a nutshell, many students performed the software-based bonus exercises and those who did so achieved also significantly better results in the classroom exercises. However, weak students (subsequently failing the course) mostly did not use the software. The reason to use the software was mainly the “extrinsic” motivation to earn additional grade points and not an “intrinsic” interest. Nevertheless, software was cited as a factor to improve interest in the course, but much less as a factor in improving understanding.

In case of RISCAL, a main deterring factor

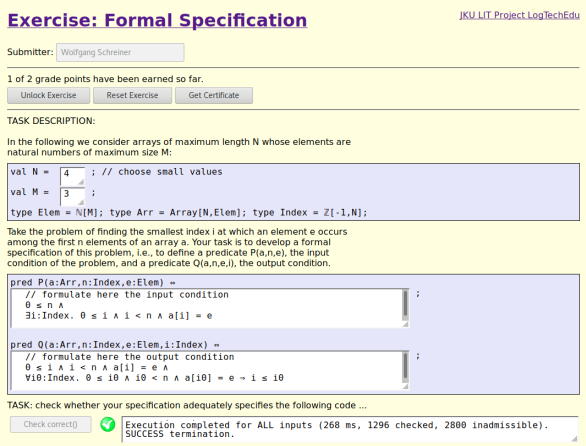

Fig. 4. A RISCAL Web Exercise

to use the software was the need for a local installation (even if by a virtual machine), the need to learn to use the software, and the need to manipulate text files. In the 2019 instance of the course, we therefore introduced a web-based frontend to a server installation of RISCAL that allowed students to perform the exercises within their web browsers (see Figure 4). Indeed, now about 170 students (from about 330 active students) submitted RISCAL-based exercises via the web interface; most with correct results. A preliminary statistical evaluation indicates that those who did so indeed performed in the course tests better than those who did not. Based on these results, we plan in 2020 to proceed with the RISCAL exercises via this frontend.

d) Correct Program and Algorithm Development: There is one kind of courses where the potential of RISCAL has not yet been applied: those on the development of programs (respectively algorithms) where their correctness with respect to given specifications should be checked. Here lecturers might hand out program assignments in the usual way by the desired interface of a procedure and an informal explanation of the inputs it can expect and the results it must deliver. However, additionally, the procedure would be equipped with a formal contract, against which the student could fully automatically check the correctness of her solution and the lecturer could fully automatically check the correctness of a submission. If a solution fails the check, the reported error also demonstrates a concrete input/output pair that demonstrates the failure. In this way, in particular, erroneous boundary cases (that very often give problems) can be quickly detected.

We are not in charge of a corresponding course where RISCAL can be applied in this sense but plan in the near future to approach other lecturers teaching program/algorithm development that may find interest in the use of RISCAL.

5. Conclusions

From the presented experience, we have good evidence that by the use of a "mathematical model checker" such as RISCAL the education in various areas of science and engineering may be substantially improved. This of course mainly applies to closely related "formal" topics, but may be also relevant for topics like programming, where RISCAL can check in small domains the correctness of programs with respect to their specifications.

Our main success lies so far on levels of education where students have already at least some prior (technical and/or formal) background. In courses targeted to absolute beginners, mainly the stronger students profit from the software, while the weaker students (already struggling with the basic material) are potentially overwhelmed by the additional "burden" to use software. Here the use of software requires careful evaluation and fine-tuning.

RISCAL itself has reached a stable state and has been applied in various courses, mostly but not only at JKU. However, while RISCAL is primarily targeted to educational scenarios that typically focus on small models, its semantics-based checking mechanism is efficient enough to also analyze models of non-trivial size. Furthermore, the novel SMT-based decision mechanism presented in this paper allows to check models of sizes that were out of the reach of RISCAL so far.

Further work will focus on the improvement of feedback mechanisms to help students understand the computed results and the integration with proof-based environments to let RISCAL be used as a "pre-checker" of formalizations in small domains before turning to general proofs in domains of arbitrary size. We also plan to extend the recently introduced notion of "concurrent systems" by a model checker for specifications in linear temporal logic (LTL).

References

[1] Barrett, C., Fontaine, P., and Tinelli, C. The Satisfiability Modulo Theories Library (SMT-LIB). http://www.SMT-LIB.org, 2016.

[2] Clarke, E. M., Henzinger, T. A., Veith, H., and Bloem, R., Eds. *Handbook of Model Checking*. Springer, Berlin, Germany, 2018. doi:10.1007/978-3-319-10575-8.

[3] Dutertre, B. Yices 2.2. In *Computer-Aided Verification (CAV'2014)* (July 2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 737–744. https://doi.org/10.1007/978-3-319-08867-9_49.

[4] Jackson, D. *Software Abstractions — Logic, Language, and Analysis*, revised ed. MIT Press, Cambridge, MA, USA, 2012. https://mitpress.mit.edu/books/software-abstractions-revised-edition.

[5] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman, Amsterdam, The Netherlands, 2002. http://research.microsoft.com/users/lamport/tla/book.html.

[6] Leino, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic Programming and Automated Reasoning (LPAR-16), Dakar, Senegal, April 25–May 1, 2010* (2010), E. M. Clarke and A. Voronkov, Eds., vol. 6355 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, pp. 348–370. doi:10.1007/978-3-642-17511-4_20.

[7] LogTechEdu. JKU LIT Project LOGTECHEDU, July 2019. http://fmv.jku.at/logtechedu.

[8] Merz, S., and Vanzetto, H. Encoding TLA+ into Many-Sorted First-Order Logic. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016* (Cham, 2016), M. J. Butler, K.-D. Schewe, A. Mashkoor, and M. Biró, Eds., vol. 9675 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 54–69. doi:10.1007/978-3-319-33600-8_3.

[9] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, Berlin, Germany, October 2017. http://isabelle.in.tum.de/doc/tutorial.pdf.

[10] Reichl, F.-X. The Integration of SMT Solvers into the RISCAL Model Checker. Master's thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, April 2020. https://www.risc.jku.at/publications/download/risc_6103/Thesis.pdf.

[11] RISCAL. The RISC Algorithm Language (RISCAL), March 2017. https://www.risc.jku.at/research/formal/software/RISCAL.

[12] Schreiner, W. The RISC Algorithm Language (RISCAL) — Tutorial and Reference Manual (Version 1.0). Technical report, RISC, Johannes Kepler University, Linz, Austria, March 2017. Available at [11].

[13] Schreiner, W. Validating Mathematical Theories and Algorithms with RISCAL. In *CICM 2018, 11th Conference on Intelligent Computer Mathematics, Hagenberg, Austria, August 13–17* (2018), F. Rabe, W. Farmer, G. Passmore, and A. Youssef, Eds., vol. 11006 of *Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence*, Springer, Berlin, pp. 248–254. doi:10.1007/978-3-319-96812-4_21.

[14] Schreiner, W. Logic and Semantic Technologies for Computer Science Education. In *Informatics'2019, 2019 IEEE 15th International Scientific Conference on Informatics* (Poprad, Slovakia, November 20–22, 2019), IEEE, pp. 7–12. To appear.

[15] Schreiner, W. Theorem and Algorithm Checking for Courses on Logic and Formal Methods. In *Post-Proceedings ThEdu'18, Theorem proving components for Educational software, Oxford, United Kingdom, July 18, 2018* (2019), P. Quaresma and W. Neuper, Eds., vol. 290 of *EPTCS*, pp. 56–75. doi:10.4204/EPTCS.290.5.

[16] Schreiner, W., Brunhuemer, A., and 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, Gothenburg, Sweden, August 6, 2017* (2018), P. Quaresma and W. Neuper, Eds., vol. 267 of *EPTCS*, pp. 120–139. doi:10.4204/EPTCS.267.8.

[17] SemTech. SemTech — Semantic Technologies for Computer Science Education, January 2018. https://www.risc.jku.at/projects/SemTech.

[18] Steingartner, W., Eldojali, M. A. M., Radakovic, D., and Dostál, J. Software support for course in Semantics of programming languages. In *IEEE 14th International Scientific Conference on Informatics* (Poprad, Slovakia, November 14–16, 2017), pp. 359–364. doi:10.1109/INFORMATICS.2017.8327275.

[19] Steingartner, W., and Novitzká, V. Learning tools in course on semantics of programming languages. In *MMFT 2017 — Mathematical Modelling in Physics and Engineering* (Czestochowa, Poland, September 18–21, 2017), pp. 137–142. http://im.pcz.pl/konferencja/get.php?doc=MMFT2017_streszczenia_wykladow.pdf.

[20] Thiébaut, D. Automatic Evaluation of Computer Programs Using Moodle's Virtual Programming Lab (VPL) Plug-in. *Journal of Computing Sciences in Colleges 30*, 6 (June 2015), 145–151. http://dl.acm.org/citation.cfm?id=2753024.2753053.