TRLC Static Checker

Florian Schanda

November 7, 2023

Motivation

- TRLC check expressions are executable
- TRLC language defines certain run-time errors
  - Null dereference
  - Division by zero
  - Array out-of-bounds access
  - Arithmetic over- and underflow[1]
- The Python reference implementation is safe (i.e. throws errors)
- A fast C implementation might not be

[1] Not checked (yet) because reference implementation uses arbitrary precision arithmetic.

Example

```
type Requirement {
  description      optional String
  safety_relevant           Boolean
}

checks Requirement {
  len(description) >= 10, "description too short"
}
```

Example

```
type Requirement {
  description      optional String
  safety_relevant           Boolean
}

checks Requirement {
  len(description) >= 10, "description too short"
}
```

```
$ trlc.py --verify foo.rsl
len(description) >= 10, "description too short"
    ^^^^^^^^^^^ foo.rsl:9: issue: expression could be null [vcg-evaluation-of-null]
              | example record_type triggering error:
              |   Requirement bad_potato {
              |     /* description is null */
              |     safety_relevant = false
              |   }
```

Is this really so hard?

- This seems obvious enough...
- We could detect that and say “always prefix with implies”

Is this really so hard?

- This seems obvious enough...
- We could detect that and say “always prefix with implies”
- But it gets tricky complex quickly

```
type Requirement {
  top_level             Boolean
  description optional  String
}

checks Requirement {
  top_level implies description != null,
    "top level requirements need a description"
}

type Top_Level_Requirement extends Requirement {
  freeze top_level = true
}

checks Top_Level_Requirement {
  len(description) >= 10, "too short"
}
```

Any complicated semantics can surprising

```
Top_Level_Requirement bad_potato {}
```

Will produce:

```
$ trlc.py foo.rsl foo.trlc
Top_Level_Requirement bad_potato {
                      ^^^^^^^^^^ foo.trlc:3: check error: top level requirements
                                             need a description
Top_Level_Requirement bad_potato {
                      ^^^^^^^^^^ foo.trlc:3: error: input to unary expression
                                             len(description) (foo.rsl:18) must not be null
```

Any complicated semantics can surprising

```
Top_Level_Requirement bad_potato {}
```

Will produce:

```
$ trlc.py foo.rsl foo.trlc
Top_Level_Requirement bad_potato {
                      ^^^^^^^^^^ foo.trlc:3: check error: top level requirements
                                             need a description
Top_Level_Requirement bad_potato {
                      ^^^^^^^^^^ foo.trlc:3: error: input to unary expression
                                             len(description) (foo.rsl:18) must not be null
```

Should have made the first check a fatal check to prevent execution!

Linter

What is it

- Formal verification tool
- Covers all possible inputs, for all possible TRLC implementations
- Models TRLC types and semantics in SMTLIB
- Generates counter-examples or proofs of absence of run-time errors

Terminology

SAT	(Boolean) satisfiability problem (NP-hard)
NP-hard	Problem where checking a solution is fast but computing a solution is non-polynomial in complexity (e.g. $O(2^n)$)
Undecidable	Problem that cannot be solved by any algorithm
SMT	SAT modulo theory, an extension of SAT with theories like integer or float arithmetic
SMT Solver	Tool to automatically solve SMT problems
SMTLIB	Language to describe problems to an SMT solver
VC	Verification condition (problem you need to solve to demonstrate something, e.g. absence of run-time errors)

Terminology II

Sound	Reasoning that does not miss bugs (i.e. no false negatives)
Complete	Reasoning that does not have false alarms
Automatic	Reasoning that does not require human intelligence as input
Over-approximate	Analysis that is sound and automatic
Under-approximate	Analysis that is complete and automatic
Deductive	Analysis that is sound and complete

Linter

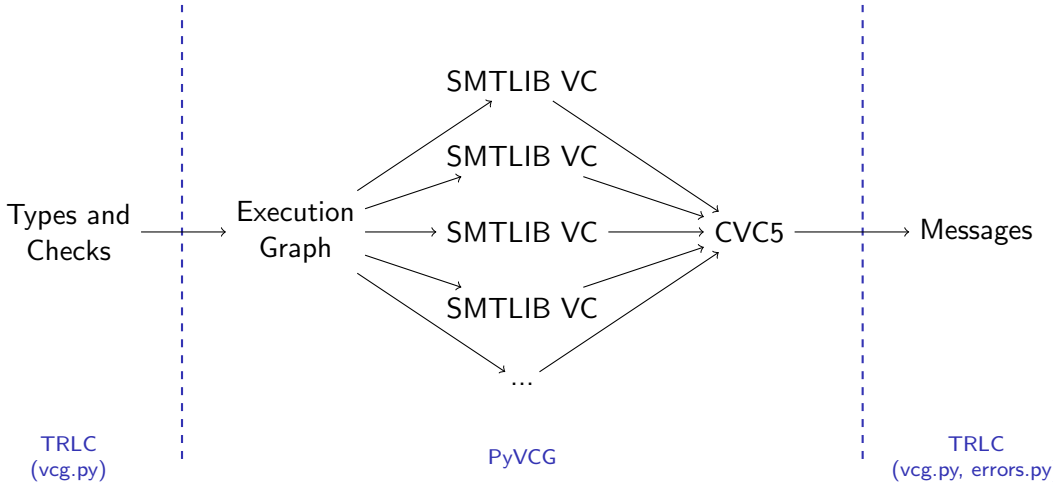
Building blocks

This would be impossible to just build from scratch, so we use tools:

- PyVCG (a low-level verification condition generator, built for TRLC initially but could be useful elsewhere)
- CVC5 (a state of the art SMT solver)

Linter

Dataflow



SMTLIB

Some examples

```
(set-logic QF_NIA)
(set-option :produce-models true)

(declare-const a      Int)
(declare-const b      Int)
(declare-const result Int)

(assert (= result
           (* a b)))

(check-sat)
(get-value (a))
(get-value (b))
(get-value (result))
```

SMTLIB

Some examples

```
(set-logic QF_NIA)
(set-option :produce-models true)

(declare-const a      Int)
(declare-const b      Int)
(declare-const result Int)

(assert (= result
           (* a b)))

(check-sat)
(get-value (a))
(get-value (b))
(get-value (result))
```

```
sat
((a 0))
((b 0))
((result 0))
```

SMTLIB

Some examples

```
(set-logic QF_NIA)
(set-option :produce-models true)

(declare-const a      Int)
(declare-const b      Int)
(declare-const result Int)

(assert (= result
           (* a b)))
(assert (= result 42))

(check-sat)
(get-value (a))
(get-value (b))
(get-value (result))
```

SMTLIB

Some examples

```
(set-logic QF_NIA)
(set-option :produce-models true)

(declare-const a      Int)
(declare-const b      Int)
(declare-const result Int)

(assert (= result
           (* a b)))
(assert (= result 42))

(check-sat)
(get-value (a))
(get-value (b))
(get-value (result))
```

```
sat
((a (- 2)))
((b (- 21)))
((result 42))
```

SMTLIB

Some examples

```
(set-logic QF_NIA)
(set-option :produce-models true)

(declare-const a      Int)
(declare-const b      Int)
(declare-const result Int)

(assert (= result
           (* a b)))
(assert (= result 42))
(assert (= a 4))

(check-sat)
(get-value (a))
(get-value (b))
(get-value (result))
```

SMTLIB

Some examples

```
(set-logic QF_NIA)
(set-option :produce-models true)

(declare-const a      Int)
(declare-const b      Int)
(declare-const result Int)

(assert (= result
           (* a b)))
(assert (= result 42))
(assert (= a 4))

(check-sat)
(get-value (a))
(get-value (b))
(get-value (result))
```

```
unsat
```

SMTLIB

How to prove something

Lets say we want to prove that $x + 1 > x$:

- First declare variables:

```
(declare-const x Int)
```

SMTLIB

How to prove something

Lets say we want to prove that $x + 1 > x$:

- First declare variables:

```
(declare-const x Int)
```

- Then define a goal:

```
(define-const goal Bool
  (> (+ x 1)
     x))
```

SMTLIB

How to prove something

Lets say we want to prove that $x + 1 > x$:

- First declare variables:

```
(declare-const x Int)
```

- Then define a goal:

```
(define-const goal Bool
  (> (+ x 1)
     x))
```

- Then assert that the goal is not true:

```
(assert (not goal))
```

- Then ask for a model:

```
(check-sat)
```

SMTLIB

How to prove something

Lets say we want to prove that $x + 1 > x$:

- First declare variables:

```
(declare-const x Int)
```

- Then define a goal:

```
(define-const goal Bool
  (> (+ x 1)
     x))
```

- Then assert that the goal is not true:

```
(assert (not goal))
```

- Then ask for a model:

```
(check-sat)
```

- If we get a model: we know it's not (always) true and we have a specific counter-example

SMTLIB

How to prove something

Lets say we want to prove that $x + 1 > x$:

- First declare variables:

```
(declare-const x Int)
```

- Then define a goal:

```
(define-const goal Bool
  (> (+ x 1)
     x))
```

- Then assert that the goal is not true:

```
(assert (not goal))
```

- Then ask for a model:

```
(check-sat)
```

- If we get a model: we know it's not (always) true and we have a specific counter-example
- If we don't: we know there are no counter-examples, i.e. the original goal is always true

TRLC

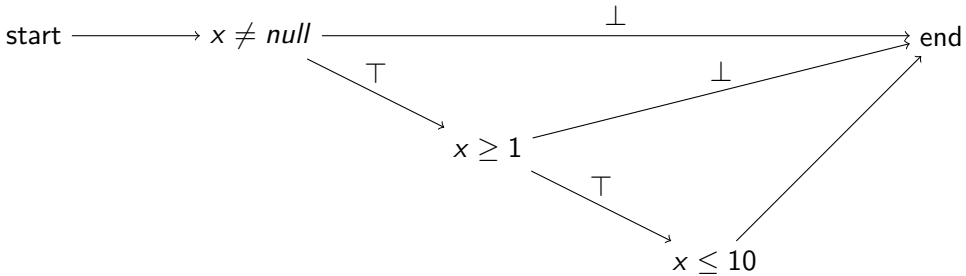
Execution semantics

- Mostly just expressions (e.g. $len(x) + len(y) > 10$)
- Control flow is rare, but we have some:
  - and, or, and implies (short-circuit semantics)
  - range tests
  - if expressions
  - ordering of (fatal) checks inside a block
  - checks from parent types before checks from extension
- Interesting cases:
  - Execution order from checks from different blocks is unspecified
  - Execution order inside quantifiers is unspecified
  - Execution continues after (non-fatal) errors and warnings

TRLC

Execution semantics example

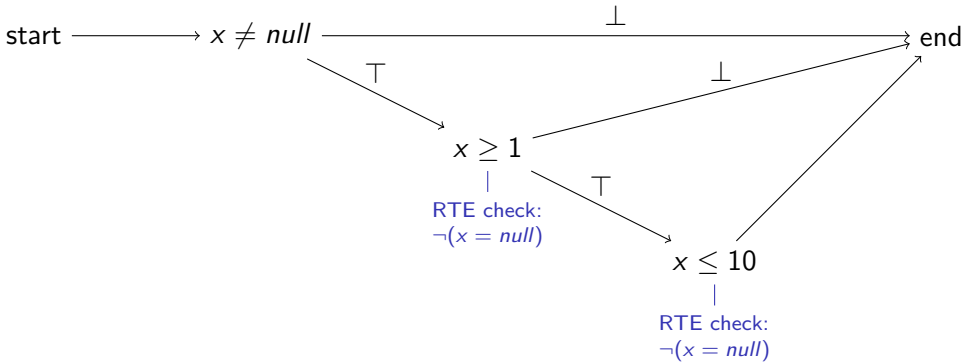
```
x != null implies x in 1 .. 10, "potato"
```



TRLC

Execution semantics example

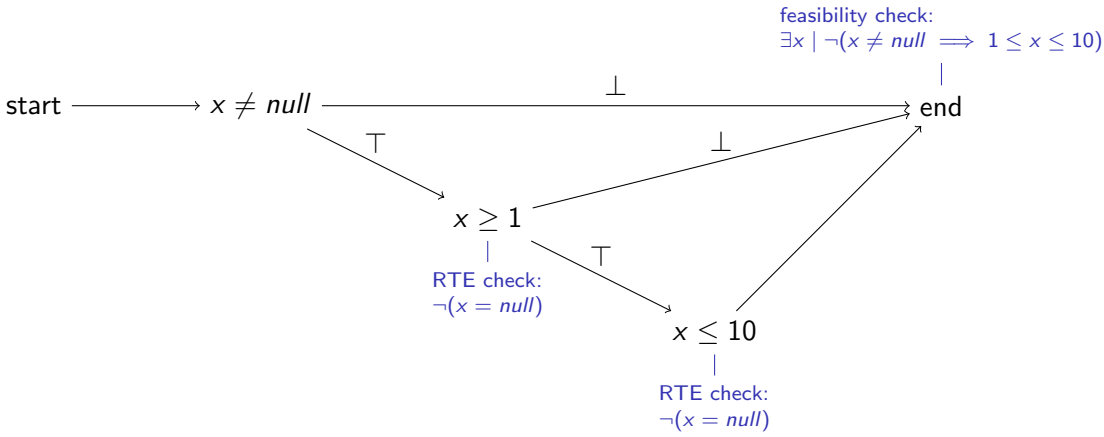
```
x != null implies x in 1 .. 10, "potato"
```



TRLC

Execution semantics example

```
x != null implies x in 1 .. 10, "potato"
```



TRLC

Required tasks

- Build execution graph and model TRLC (TRLC with PyVCG)
- Annotate this graph with checks (TRLC with PyVCG)
- Generate VCs (PyVCG)
- Solve them (PyVCG with CVC5)
- Generate feedback to the user (TRLC with PyVCG)

TRLC

PyVCG

- PyVCG does all the graph and SMT stuff
- I decided to factor it out since it could be reusable for other projects
- I decided not to use Why3[2] or Boogie[3] because I don't like quantifiers
- I decided not to use GOTO[4] because its somewhat annoying to work with and I *do* need quantifiers and infinite integers
- (Also, I was extremely bored)

[2] https://why3.lri.fr
[3] https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language
[4] https://www.cprover.org/cbmc/

Modelling TRLC

Why

TRLC $\neq$ SMTLIB:

- TRLC is sequential and executable
- SMTLIB is declarative
- Types are different
- There are no “null” values in SMTLIB

Modelling TRLC

Null

- Optional components can be “null”
- Expressions generally can’t be (`x + 1` itself cannot be null, but `x` could be)
- Quantification variables can’t be (in `(forall x in y => x)` the bound variable `x` cannot be null, but `y` could be)
- Array members can’t be (although the whole array itself could be)

Modelling TRLC

Null

I considered two options:

- Make a datatype for everything: $x = (isNull, actualValueOfX)$
  - \+ Very generic, easy to get right
  - \- Requires re-implementation of all operations
  - \- Has the smell of three-valued logic
  - \- Might make quantifiers worse
- Make a separate value for everything: $x = actualValueofX, valid\_x$
  - \- Requires more manual work when generating VCs
  - \+ Probably way faster if you can optimise it away

I chose the second option.

Modelling TRLC

Null

How this looks like with $x = y$:

Modelling TRLC

Null

How this looks like with $x = y$:

- There are four variables here:
  - x and y
  - $valid\_x$ and $valid\_y$

Modelling TRLC

Null

How this looks like with $x = y$:

- There are four variables here:
  - x and y
  - $valid\_x$ and $valid\_y$
- Semantics of equality are:

  Equality_On_Null Null is only equal to itself.

  Simple_Relational_Semantics The meaning of the relationship operators are the usual.

Modelling TRLC

Null

How this looks like with $x = y$:

- There are four variables here:
  - x and y
  - $valid\_x$ and $valid\_y$
- Semantics of equality are:

  Equality_On_Null Null is only equal to itself.

  Simple_Relational_Semantics The meaning of the relationship operators are the usual.
- $valid\_x = valid\_y \wedge (valid\_x \implies x = y)$

Modelling TRLC

Null

- So $x = y$ is $valid\_x = valid\_y \wedge (valid\_x \implies x = y)$

Modelling TRLC

Null

- So $x = y$ is $valid\_x = valid\_y \wedge (valid\_x \implies x = y)$
- If we know that y is not optional, we could simplify:
  - Flip around: $valid\_y = valid\_x \wedge (valid\_y \implies y = x)$

Modelling TRLC

Null

- So $x = y$ is $valid\_x = valid\_y \wedge (valid\_x \implies x = y)$
- If we know that y is not optional, we could simplify:
  - Flip around: $valid\_y = valid\_x \wedge (valid\_y \implies y = x)$
  - Substitute $\top$ for $valid\_y$: $\top = valid\_x \wedge (\top \implies y = x)$

Modelling TRLC

Null

- So $x = y$ is $valid\_x = valid\_y \wedge (valid\_x \implies x = y)$
- If we know that y is not optional, we could simplify:
  - Flip around: $valid\_y = valid\_x \wedge (valid\_y \implies y = x)$
  - Substitute $\top$ for $valid\_y$: $\top = valid\_x \wedge (\top \implies y = x)$
  - Simplify: $valid\_x \wedge y = x$

Modelling TRLC

Null

- So $x = y$ is $valid\_x = valid\_y \wedge (valid\_x \implies x = y)$
- If we know that y is not optional, we could simplify:
  - Flip around: $valid\_y = valid\_x \wedge (valid\_y \implies y = x)$
  - Substitute $\top$ for $valid\_y$: $\top = valid\_x \wedge (\top \implies y = x)$
  - Simplify: $valid\_x \wedge y = x$
- If we know that x is also not optional, we could simplify further: $x = y$

Modelling TRLC

Null

Final strategy:

- Introduce a new Boolean for each value indicating the validity
- Perform arithmetic on just the values, assuming validity is OK
- Perform validity checks on just the validity values
- Only place we mix is for the == and != operators
- Most of it can be simplified

Modelling TRLC

Null

A real example:

```
type T {
  x optional Boolean
  y optional Boolean
}
checks T {
  x == y, "potato"
}
```

Modelling TRLC

Null

A real example:

```
;; value for x declared on foo.rsl:3:3
(declare-const |Foo.T.x.value| Bool)
(declare-const |Foo.T.x.valid| Bool)
;; value for y declared on foo.rsl:4:3
(declare-const |Foo.T.y.value| Bool)
(declare-const |Foo.T.y.valid| Bool)
;; result of x == y at foo.rsl:7:5
(define-const |tmp.1| Bool
  (and (= |Foo.T.x.valid| |Foo.T.y.valid|)
       (=> |Foo.T.x.valid| (= |Foo.T.x.value| |Foo.T.y.value|))))
```

Modelling TRLC

Execution graph and null

One more example:

```
type T {
  x optional Integer
}
checks T {
  x + 1 > x, "potato"
}
```

Modelling TRLC

Execution graph and null

One more example:

```
type T {
  x optional Integer
}
checks T {
  x + 1 > x, "potato"
}
```

Requires more than one VC:

- VC1: Validity check for x on LHS
- We can then compute the value of $x + 1$ assuming it's valid

Modelling TRLC

Execution graph and null

One more example:

```
type T {
  x optional Integer
}
checks T {
  x + 1 > x, "potato"
}
```

Requires more than one VC:

- VC1: Validity check for x on LHS
- We can then compute the value of $x + 1$ assuming it's valid
- VC2: Validity check for x on RHS
- We can then compute the value of $x + 1 > x$ assuming it's valid

Modelling TRLC

Execution graph and null

One more example:

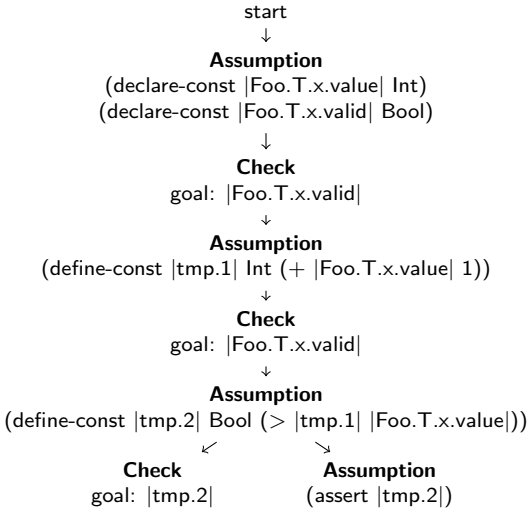
```
type T {
  x optional Integer
}
checks T {
  x + 1 > x, "potato"
}
```

Requires more than one VC:

- VC1: Validity check for x on LHS
- We can then compute the value of $x + 1$ assuming it's valid
- VC2: Validity check for x on RHS
- We can then compute the value of $x + 1 > x$ assuming it's valid
- We can now use the computed values to do something like check if it's always true

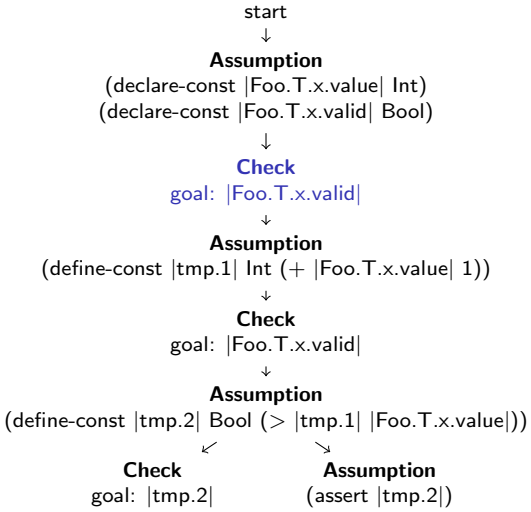
Modelling TRLC

Execution graph and null



Modelling TRLC

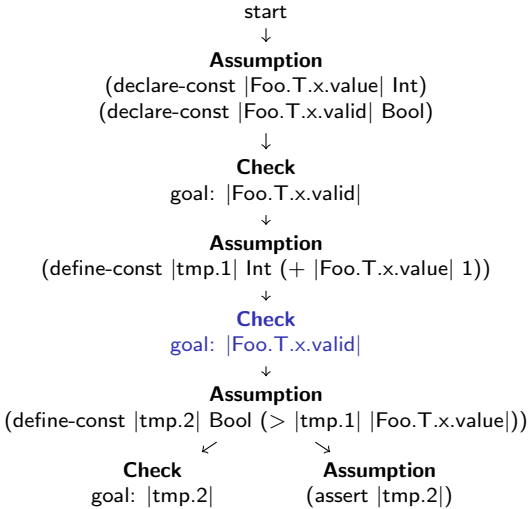
Execution graph and null




```
;; value for x declared on foo.rsl:3:3
(declare-const |Foo.T.x.value| Int)
(declare-const |Foo.T.x.valid| Bool)
;; validity check for x
(assert (not |Foo.T.x.valid|))
(check-sat)
```

Modelling TRLC

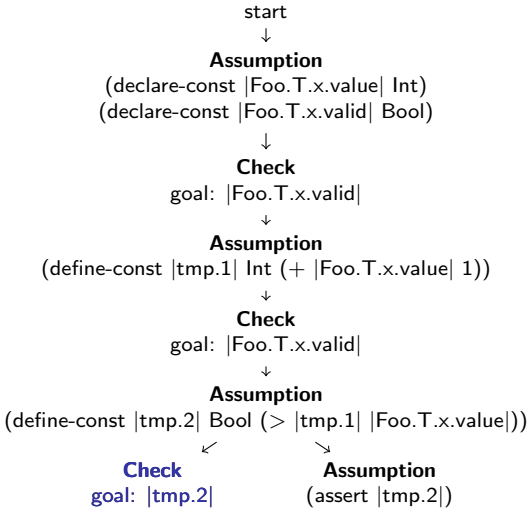
Execution graph and null




```
;; value for x declared on foo.rsl:3:3
(declare-const |Foo.T.x.value| Int)
(declare-const |Foo.T.x.valid| Bool)
(assert |Foo.T.x.valid|)
;; result of x + 1 at foo.rsl:6:5
(define-const |tmp.1| Int
  (+ |Foo.T.x.value| 1))
;; validity check for x
(assert (not |Foo.T.x.valid|))
(check-sat)
```

Modelling TRLC

Execution graph and null




```
;; value for x declared on foo.rsl:3:3
(declare-const |Foo.T.x.value| Int)
(declare-const |Foo.T.x.valid| Bool)
(assert |Foo.T.x.valid|)
;; result of x + 1 at foo.rsl:6:5
(define-const |tmp.1| Int
  (+ |Foo.T.x.value| 1))
(assert |Foo.T.x.valid|)
;; result of x + 1 > x at foo.rsl:6:9
(define-const |tmp.2| Bool
  (> |tmp.1| |Foo.T.x.value|))
;; feasability check for x + 1 > x
(assert (not |tmp.2|))
(check-sat)
```

Modelling TRLC

Debugging options

You can see this for real using the `--debug-vcg` option:

- Generates a `.pdf` for the graph
- Generates a `.smt2` file for each VC

Modelling TRLC

Types

We have these types in TRLC:

- Boolean
- Integer
- Decimal
- String (and Markup_String)
- Enumerations
- User-defined records
- User-defined tuples
- Arrays consisting out of any of the above

Modelling TRLC

Types

We have these types in TRLC:

- Boolean (SMTLIB Bool)
- Integer (SMTLIB Int)
- Decimal
- String (and Markup_String)
- Enumerations
- User-defined records
- User-defined tuples
- Arrays consisting out of any of the above

Modelling TRLC

Types

We have these types in TRLC:

- Boolean (SMTLIB Bool)
- Integer (SMTLIB Int)
- Decimal (SMTLIB Real, over-approximation)
- String (and Markup_String)
- Enumerations
- User-defined records
- User-defined tuples
- Arrays consisting out of any of the above

Modelling TRLC

Types

We have these types in TRLC:

- Boolean (SMTLIB Bool)
- Integer (SMTLIB Int)
- Decimal (SMTLIB Real, over-approximation)
- String (and Markup_String) (SMTLIB String, over-approximation for Markup_String)
- Enumerations
- User-defined records
- User-defined tuples
- Arrays consisting out of any of the above

Modelling TRLC

Types

We have these types in TRLC:

- Boolean (SMTLIB Bool)
- Integer (SMTLIB Int)
- Decimal (SMTLIB Real, over-approximation)
- String (and Markup_String) (SMTLIB String, over-approximation for Markup_String)
- Enumerations (SMTLIB Datatypes)
- User-defined records
- User-defined tuples
- Arrays consisting out of any of the above

Modelling TRLC

Types

We have these types in TRLC:

- Boolean (SMTLIB Bool)
- Integer (SMTLIB Int)
- Decimal (SMTLIB Real, over-approximation)
- String (and Markup_String) (SMTLIB String, over-approximation for Markup_String)
- Enumerations (SMTLIB Datatypes)
- User-defined records (SMTLIB Datatypes)
- User-defined tuples (SMTLIB Datatypes)
- Arrays consisting out of any of the above

Modelling TRLC

Types

We have these types in TRLC:

- Boolean (SMTLIB Bool)
- Integer (SMTLIB Int)
- Decimal (SMTLIB Real, over-approximation)
- String (and Markup_String) (SMTLIB String, over-approximation for Markup_String)
- Enumerations (SMTLIB Datatypes)
- User-defined records (SMTLIB Datatypes)
- User-defined tuples (SMTLIB Datatypes)
- Arrays consisting out of any of the above (SMTLIB Sequence)

Modelling TRLC

Decimals

Decimals are annoying... There are no decimals in SMTLIB.

$$\mathbb{D} \in \mathbb{Q} \in \mathbb{R}$$

Modelling TRLC

Decimals

Decimals are annoying... There are no decimals in SMTLIB.

$$\mathbb{D} \in \mathbb{Q} \in \mathbb{R}$$

- 0.3 is a decimal

Modelling TRLC

Decimals

Decimals are annoying... There are no decimals in SMTLIB.

$$\mathbb{D} \in \mathbb{Q} \in \mathbb{R}$$

- 0.3 is a decimal
- $\frac{1}{3}$ is a rational (but not decimal)

Modelling TRLC

Decimals

Decimals are annoying... There are no decimals in SMTLIB.

$$\mathbb{D} \in \mathbb{Q} \in \mathbb{R}$$

- 0.3 is a decimal
- $\frac{1}{3}$ is a rational (but not decimal)
- $\sqrt{2}$ is a real (but not rational, and also not a decimal)

Modelling TRLC

Decimals

Decimals are annoying... There are no decimals in SMTLIB.

$$\mathbb{D} \in \mathbb{Q} \in \mathbb{R}$$

- 0.3 is a decimal
- $\frac{1}{3}$ is a rational (but not decimal)
- $\sqrt{2}$ is a real (but not rational, and also not a decimal)
- There are things true in $\mathbb{D}$ that are not true $\mathbb{R}$: $\forall x \in \mathbb{D} \mid x + \frac{1}{3} \neq 0$

Modelling TRLC

Decimals

Decimals are annoying... There are no decimals in SMTLIB.

$$\mathbb{D} \in \mathbb{Q} \in \mathbb{R}$$

- 0.3 is a decimal
- $\frac{1}{3}$ is a rational (but not decimal)
- $\sqrt{2}$ is a real (but not rational, and also not a decimal)
- There are things true in $\mathbb{D}$ that are not true $\mathbb{R}$: $\forall x \in \mathbb{D} \mid x + \frac{1}{3} \neq 0$
- There are things true in $\mathbb{R}$ that are not true $\mathbb{D}$: $\exists x \in \mathbb{R} \mid x * x = 2$

Modelling TRLC

Decimals

Options:

- Beg CVC5 developers for a Decimal extension (I tried)

Modelling TRLC

Decimals

Options:

- Beg CVC5 developers for a Decimal extension (I tried)
- Model as a pair of integers $value = \frac{a}{b}$ and say:
  - $\exists n \in \mathbb{N} | n > 0 \wedge b = 10^n$

Modelling TRLC

Decimals

Options:

- Beg CVC5 developers for a Decimal extension (I tried)
- Model as a pair of integers $value = \frac{a}{b}$ and say:
  - $\exists n \in \mathbb{N} | n > 0 \wedge b = 10^n$
- This is awful because:
  - $\mathbb{N}/\mathbb{R}$ conversions
  - power is basically unsupported
  - existential quantification
  - nonlinear division everywhere

Modelling TRLC

Decimals

Options:

- Rational with restricted precision $value = \frac{a}{b}$
  - E.g. $b \in \{1, 10, 100, 1000, 10000, 100000\}$ for 5 decimal digits
  - This is an under-approximation (i.e. not sound)
  - disjunctions everywhere

Modelling TRLC

Decimals

Options:

- Rational with restricted precision $value = \frac{a}{b}$
  - E.g. $b \in \{1, 10, 100, 1000, 10000, 100000\}$ for 5 decimal digits
  - This is an under-approximation (i.e. not sound)
  - disjunctions everywhere
- Treat as real (over-approximation)
  - Best performance
  - You sometimes get impossible counter-examples

  (This is what I chose.)

Modelling TRLC

Decimals

Example of an incorrect counter-example:

```
type T {
  a Decimal
}

checks T {
  1.0 / (a + (1.0 / 3.0)) > 0.0, "potato"
}
```

```
1.0 / (a + (1.0 / 3.0)) > 0.0, "potato"
    ^ test2.rsl:8: issue: divisor could be 0.0 [vcg-div-by-zero]
    | example record_type triggering error:
    |   T bad_potato {
    |     a = -1 / 3
    |   }
```