

How to Improve the Robustness of Auto-Active Program Proof Through Redundancy

Yannick Moy - AdaCore



How to Improve the Robustness of Auto-Active Program Proof Through Redundancy

Yannick Moy - AdaCore



Auto-Active Program Proof

Auto-Active Program Proof

Tools where user input is supplied before VC generation therefore lie between automatic and interactive verification, which we will give the name auto-active verification.

in “Usable Auto-Active Verification”, K. Rustan M. Leino and Michał Moskal

Examples of auto-active program proof:

- based on Boogie: AutoProof, Dafny, SMACK
- based on Why3: Frama-C, SPARK
- other IVL: Viper, Crucible/What4
- others: F*, OpenJML, VeriFast

Automatic Provers

Alt-Ergo

an SMT-based theorem prover supporting q from [this page](#).

Beagle

a theorem prover for first-order logic with e

CVC3

an SMT-based theorem prover; available fr

CVC4

an SMT solver supporting quantifiers and m

E prover

a theorem prover for first-order logic with e

Gappa

a solver specialized on the verification of n

Metis

a theorem prover for first order logic with e

Metitarski

a prover specialized on verification of nume

Princess

a prover for first-order logic modulo linear l

Psyche

a modular platform for automated or intera

Simplify

an automatic SMT-based prover available u

SPASS

a theorem prover for first-order logic with e

Vampire

a theorem prover for first-order logic with e

veriT

an SMT-based theorem prover supporting q

Yices

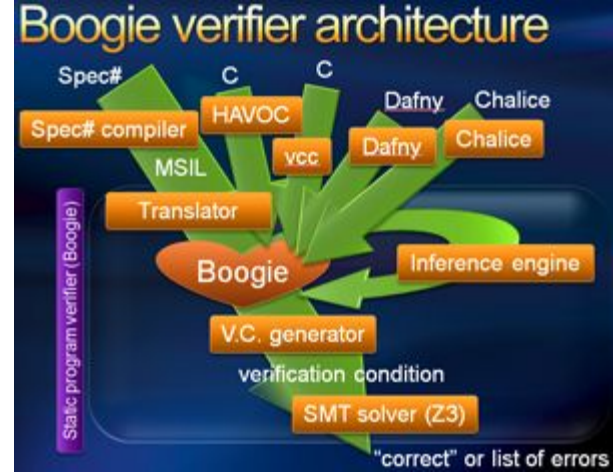
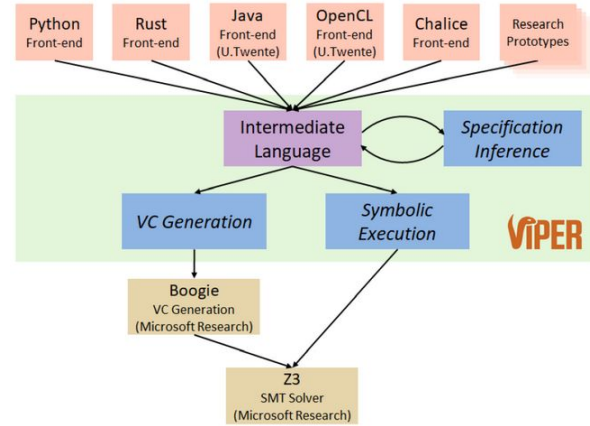
an SMT solver supporting equality, linear re

Z3

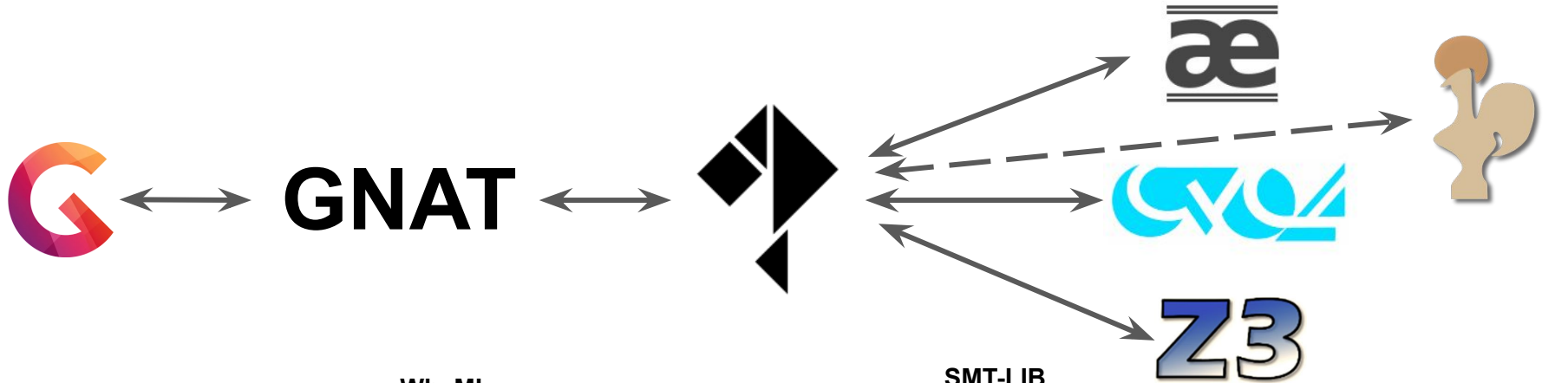
an SMT solver supporting quantifiers and m

polyprovism

monoprovism



SPARK - Auto-Active Proof for Ada Programs



WhyML

SMT-LIB

SPARK

```
A(1) := 42;
```

```
a.map__content <-  
  set  
    (a.map__content)  
  (let temp = 1 : int in  
    assert { temp ... };  
    temp)  
(42 : value)
```

```
(assert  
  (not  
    (=> (dynamic_property 0 1000000  
          (to_rep a__first) (to_rep a__last))  
        (=> (and (= (to_rep a__first) 1)  
                  (<= 0 (to_rep a__last))))  
          (<= (to_rep a__first) 1))))))  
(check-sat)
```

The Robustness Problem

The Fragility of Automatic Proofs

Prover can be non-deterministic in some settings

➤ run the prover in a deterministic setting

The Fragility of Automatic Proofs

Prover can be non-deterministic in some settings

- run the prover in a deterministic setting

Use of timeout is neither portable nor reproducible

- limit instead the “effort” done by the prover (aka “rlimit” in CVC4/Z3)

The Fragility of Automatic Proofs

Prover can be non-deterministic in some settings

- run the prover in a deterministic setting

Use of timeout is neither portable nor reproducible

- limit instead the “effort” done by the prover (aka “rlimit” in CVC4/Z3)

Unrelated changes in Verification Condition can lead to unprovable VC

- compute minimal context, use slicing

The Fragility of Automatic Proofs

Prover can be non-deterministic in some settings

- run the prover in a deterministic setting

Use of timeout is neither portable nor reproducible

- limit instead the “effort” done by the prover (aka “rlimit” in CVC4/Z3)

Unrelated changes in Verification Condition can lead to unprovable VC

- compute minimal context, use slicing

Proof can change from instantaneous to impossible due to:

- reordering of definitions, renaming of symbols
- code changes
- changes in VC generation
- changes in the prover

The Impact of Fragile Proofs

On tool users:

- Unsatisfied users when new tool release leads to regressions
- Puzzled users when minor code changes “lose” some proofs
- Lower the confidence of users in the tool

On tool developers:

- Difficult-to-investigate proof regressions
- High cost of maintaining a large regression testsuite

An Example - Is_Heap From SPARK-by-Example

```
type Heap is record
  A      : T_Arr (1 .. MAX_SIZE) := (others => 0);
  Size   : Natural               := 0;
end record;
```

```
function Is_Heap
(H : Heap)
return Boolean with
Post => Is_Heap'Result = Is_Heap_Def (H);
```

```
function Heap_Parent
(I : Positive)
return Positive is (I / 2) with
Pre => I > 1;
```

```
function Is_Heap_Def
(H : Heap)
return Boolean is
(H.Size <= H.A'Last
and then
(H.Size = 0
or else
(for all I in 2 .. H.Size =>
H.A (I) <= H.A (Heap_Parent (I))))); end Is_Heap;
```

```
function Is_Heap
(H : Heap)
return Boolean
is
  Parent : Natural := 1;
begin
  if H.Size > H.A'Length then
    return False;
  else
    if H.Size >= 1 then
      for Child in 2 .. H.Size loop
        if H.A (Parent) < H.A (Child) then
          return False;
        end if;

        pragma Loop_Invariant
          (1 <= Parent and then Parent < Child
          and then Child <= H.Size);
        pragma Loop_Invariant (Parent = Heap_Parent (Child));
        pragma Loop_Invariant (Is_Heap_Def ((A => H.A, Size => Child)));
        pragma Assert (if Child = H.Size then Is_Heap_Def (H));
        if Child mod 2 = 1 then
          Parent := Parent + 1;
        end if;
      end loop;
      pragma Assert (Is_Heap_Def (H));
    end if;

    return True;
  end if;
```

Provability of Is_Heap Across Versions of SPARK

All 3 provers Alt-Ergo, CVC4, Z3 prove all checks in all versions 2016..2020

... except for the postcondition of Is_Heap

... but GNATprove proves it always using all three provers!

	2016	2018	2019	2020
Alt-Ergo	Red	Red	Green	Green
CVC4	Green	Red	Red	Red
Z3	Red	Green	Red	Red
GNATprove	Green	Green	Green	Green

Provability of Is_Heap and Assertions Used

The 2 assertions in the code have no effect on the results

... but if we add the following assertion:

```
if H.A (Parent) < H.A (Child) then
  pragma Assert (Parent = Heap_Parent (Child));
  return False;
end if;
```

... then GNATprove proves Is_Heap fully using each of the three provers!

	2016	2018	2019	2020
Alt-Ergo				
CVC4				
Z3				
GNATprove				

The Experiment

The Experiment Proposal

Hypothesis:

using multiple provers and redundant assertions increases robustness

Hard to test against realistic code changes

- typical code patch includes code and spec/proof changes
- most code changes require spec/proof changes
- test instead against changes in the tool (VC generation and provers)

The Experiment Setup

3 successive versions of SPARK

3 provers: Alt-Ergo, CVC4, Z3

3 projects:

- SPARK-by-Example: collection of programs for teaching program proof
- SPARKNaCl: rewrite of the cryptographic library TweetNaCl
- SPARK Red-Black Trees: basis for NFM 2017 article

	2018	2019	2020
Alt-Ergo	1.30	2.3.0	2.3.0+
CVC4	1.6	1.7.1	1.8
Z3	4.6.0	4.8.0	4.8.6

	sloc	# pre/post/predicate	# assert	# checks (2019)
SPARK-by-Example	5984	164	65	1709
SPARKNaCl	2845	48	82	1065
Red-Black Trees	2531	75	109	2817

The Experiment Details

Impossible to track individual checks between SPARK versions:

- changes in “trivial” checks not sent to provers
- different locations for messages on proved and unproved checks

Combination of provers is more than combination of their results: checks in the form of conjunctions (including universally quantified and conditional expressions) treated as separate VCs

■■■➔ Run GNATprove once for each combination of provers

Scripts and results available at <https://github.com/yannickmoy/SPARKrobustX>

A Note on Reproducibility

Reproducible GNATprove runs require using prover “steps” (aka “rlimit”)
... very volatile from version to version,
... from prover to prover,
... and from VC to VC

Here we use instead “timeouts” as a measure of effort

We chose 1 minute timeout for every individual proof on a VC

In most cases, a good approximation of the results with 10 or even ∞ minutes

The Results

SPARK-by-Example

<https://github.com/tofgarion/spark-by-example>

Developed by researcher Christophe Garion and his students from ISAE-SUPAERO for teaching

All proved with SPARK Community 2018 (using maximum timeout 20 seconds) except a few checks intentionally showing the need for ghost code

Partially migrated to SPARK Community 2019

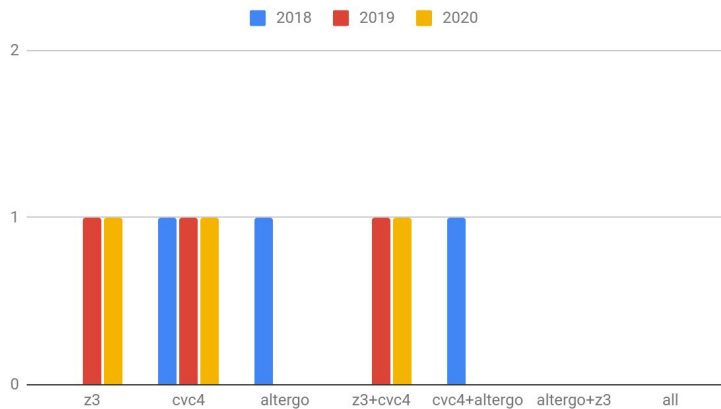
SPARK-by-Example - Is_Heap

Reminder:

	2016	2018	2019	2020
Alt-Ergo	Red	Red	Green	Green
CVC4	Green	Red	Red	Red
Z3	Red	Green	Red	Red
GNATprove	Green	Green	Green	Green

2018, 2019 and 2020

Now displayed:



SPARK-by-Example - Heap Algorithms

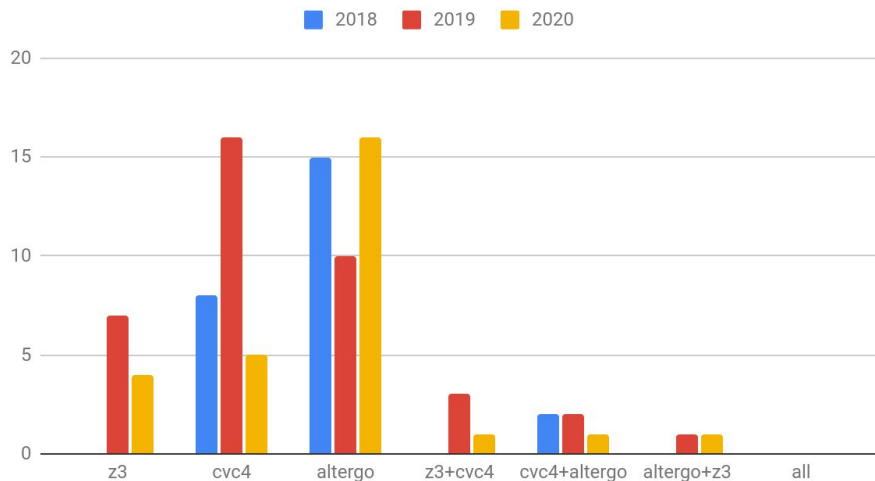
Large variations between versions for individual provers

Much smaller variations (in absolute numbers) with prover combinations

Not much difference between runs with/without assertions

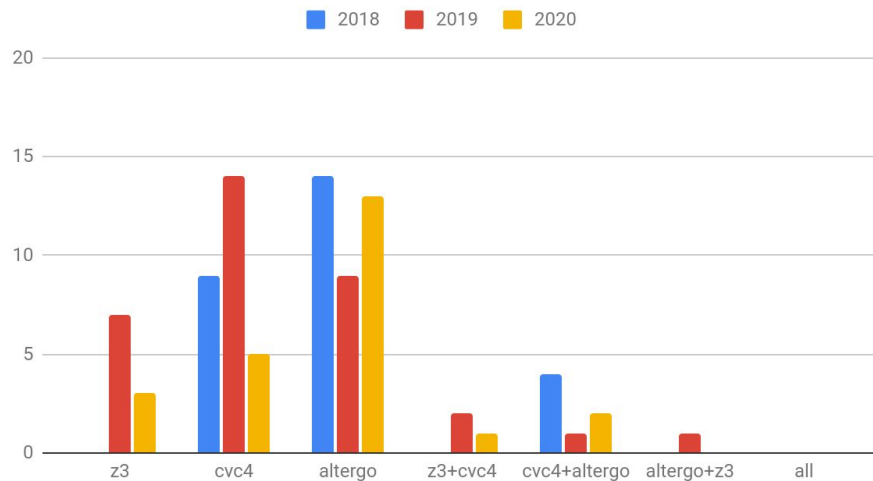
with assertions:

2018, 2019 and 2020



without assertions:

2018, 2019 and 2020



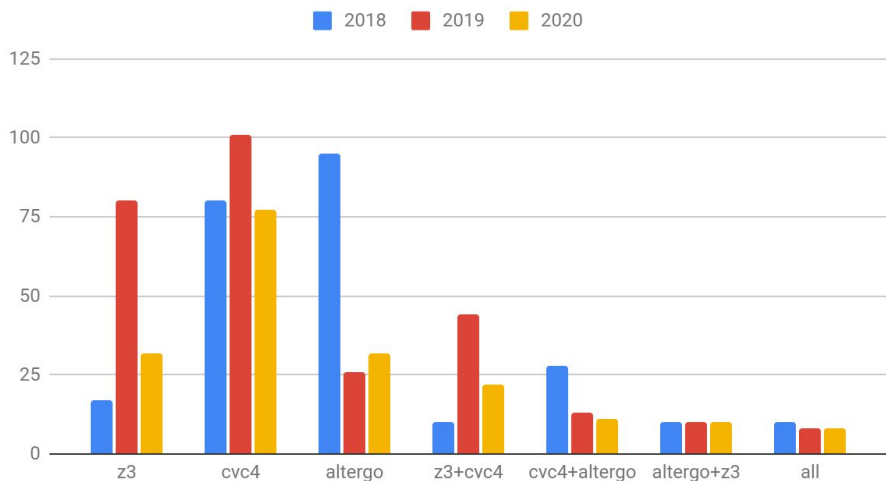
SPARK-by-Example - Unproved Checks

Same remarks as on heap algorithms

With all provers together, +80/137% more unproved checks without assertions

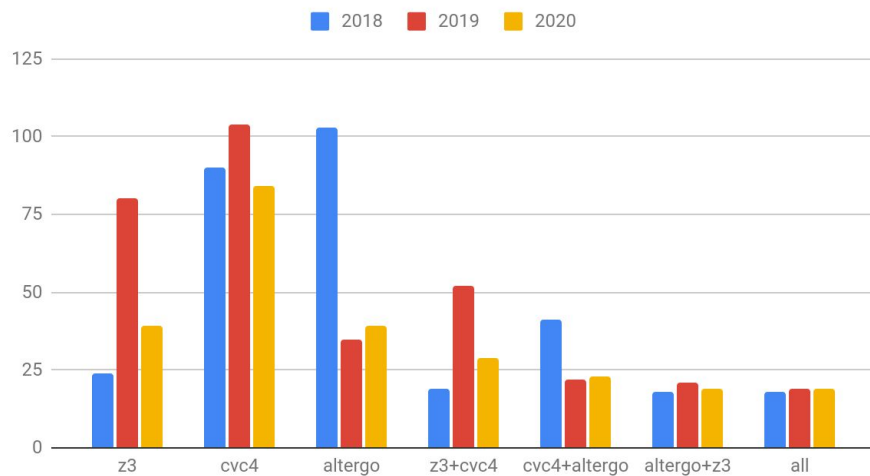
with assertions:

2018, 2019 and 2020



without assertions:

2018, 2019 and 2020

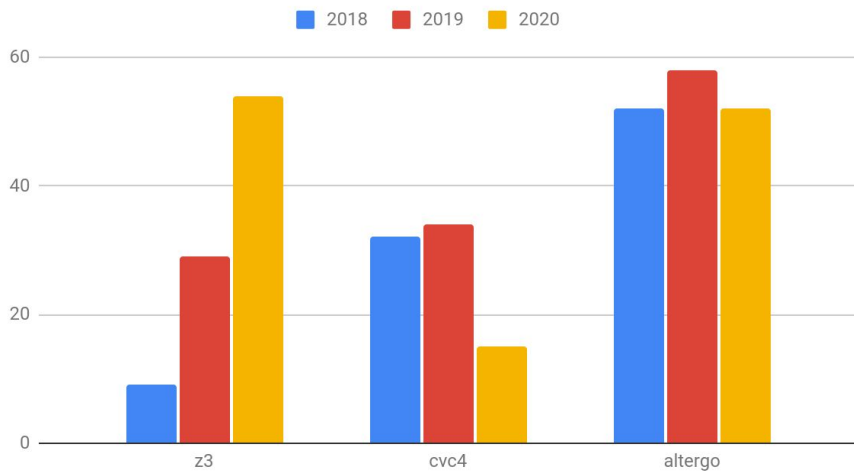


SPARK-by-Example - Max Time for Proved Checks

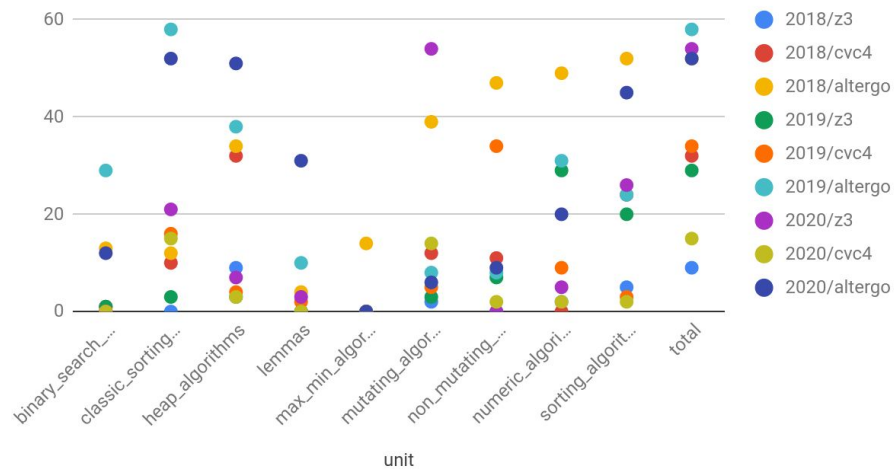
Maximum time of proof not (inversely) correlated with robustness

No explanation for 3 times as many unproved for Z3 in 2019, for Alt-Ergo in 2018

2018, 2019 and 2020



maximum time for proved checks



SPARKNaCl

<https://github.com/rod-chapman/SPARKNaCl>

Developed by Rod Chapman

All proved with SPARK Community 2019 (using steps limit)

Migrated to SPARK Community 2020

We use here an updated version of the one for SPARK Community 2019 to make it acceptable for SPARK Community 2018

SPARKNaCl - Unproved Checks

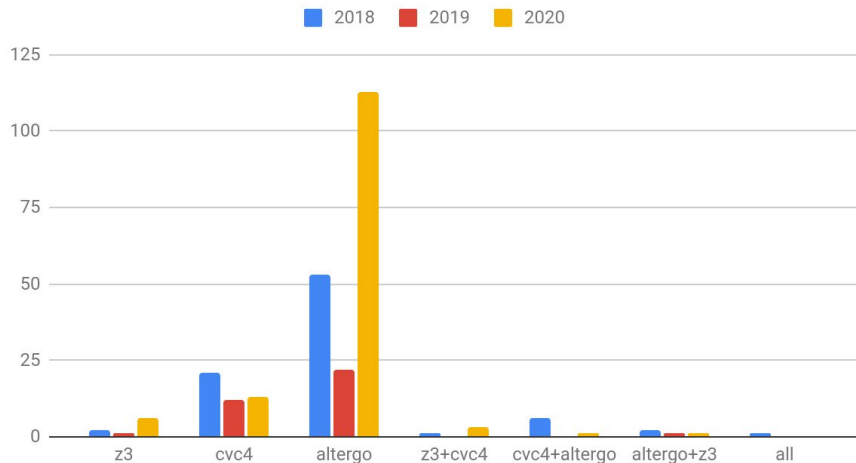
Good results of Z3 are the main driver for overall results

Alt-Ergo + CVC4 results very good despite much larger unproved checks of each

“U” shape for all individual provers \Rightarrow typical of robustness issues

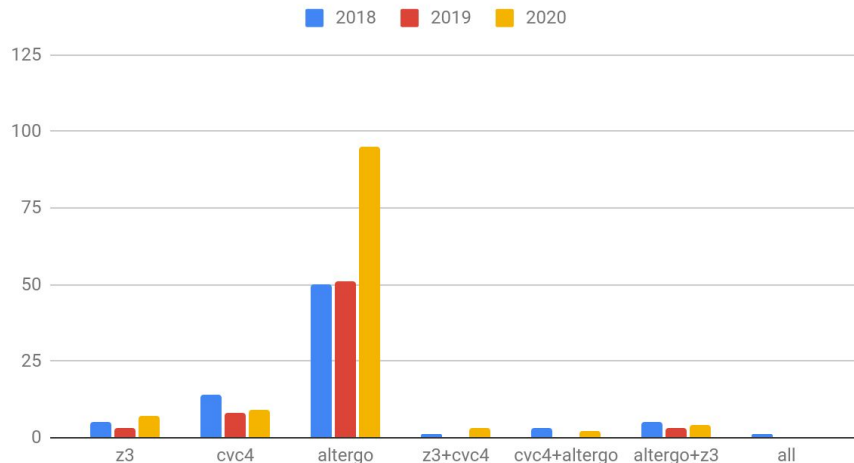
with assertions:

2018, 2019 and 2020



without assertions:

2018, 2019 and 2020

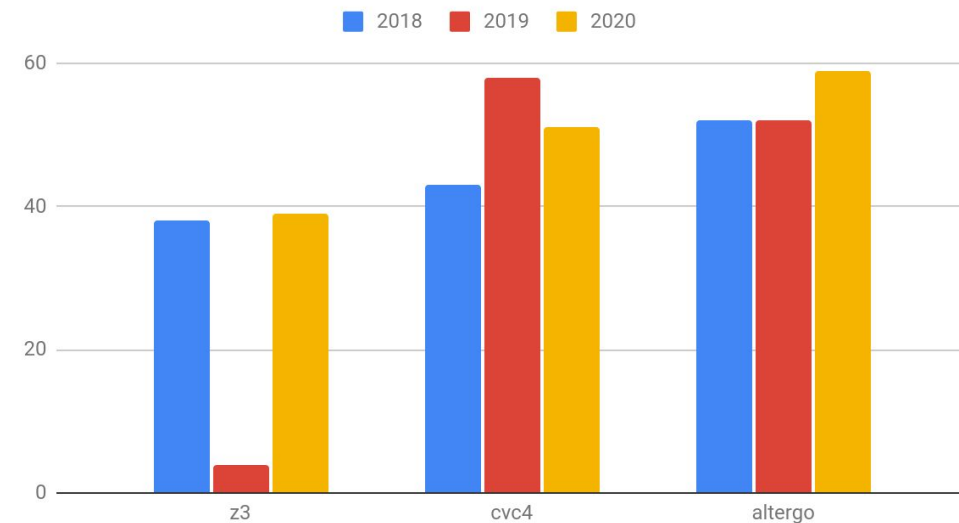


SPARKNaCl - Max Time for Proved Checks

Z3 max time always far from 60 seconds timeout

Correlates here with good results of Z3

2018, 2019 and 2020



SPARK Red-Black Trees

http://toccata.lri.fr/gallery/spark_red_black_trees.en.html

Developed by Claire Dross

Initially all proved with SPARK Pro 2017

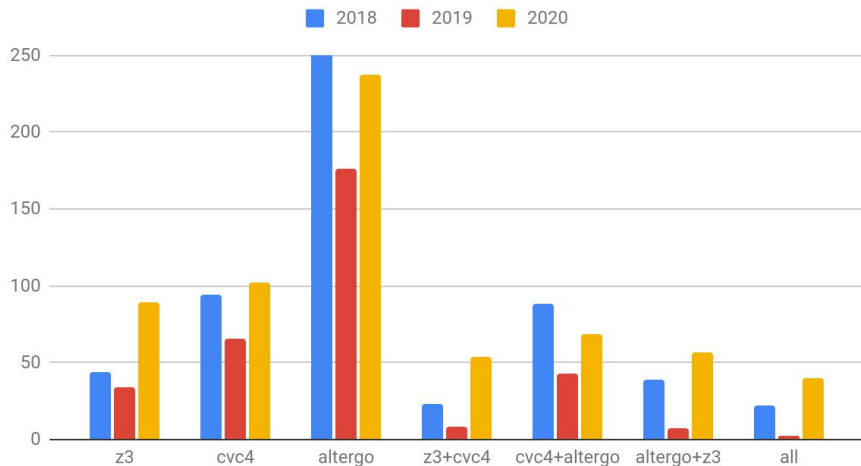
All proved with SPARK Community 2019 (using replay or larger timeout)

SPARK Red-Black Trees - Unproved Checks

marked “U” shape for all individual provers \Rightarrow clear robustness issues
more visible even with assertions as more checks to prove
prover combination always improves results

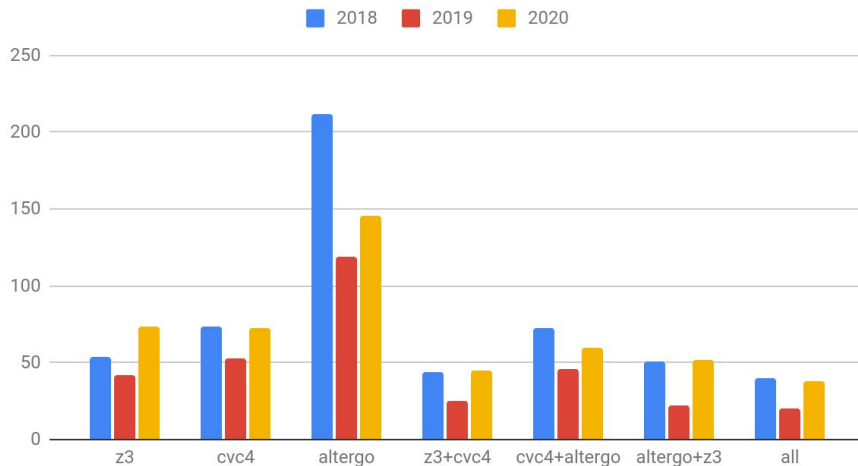
with assertions:

2018, 2019 and 2020



without assertions:

2018, 2019 and 2020

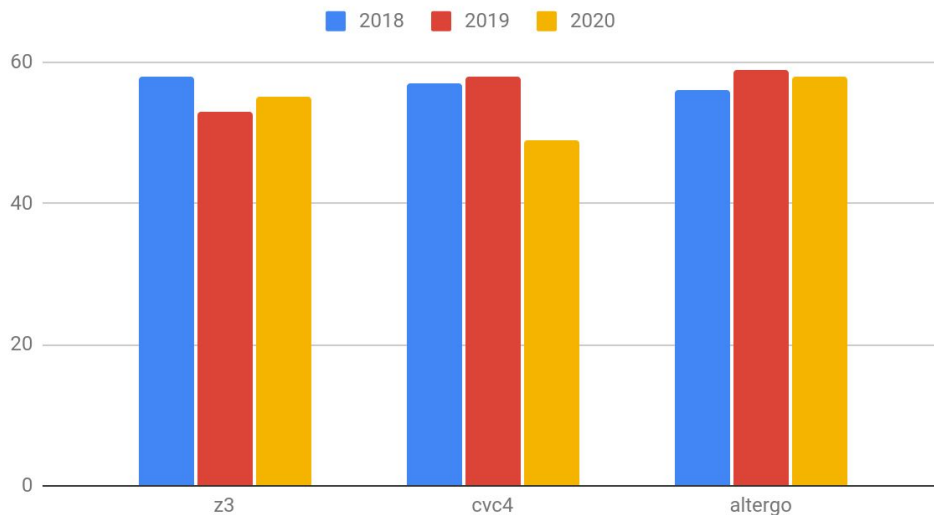


SPARK Red-Black Trees - Max Time for Proved Checks

Max time for all provers close to 60 seconds timeout

Correlates here with robustness issues

2018, 2019 and 2020



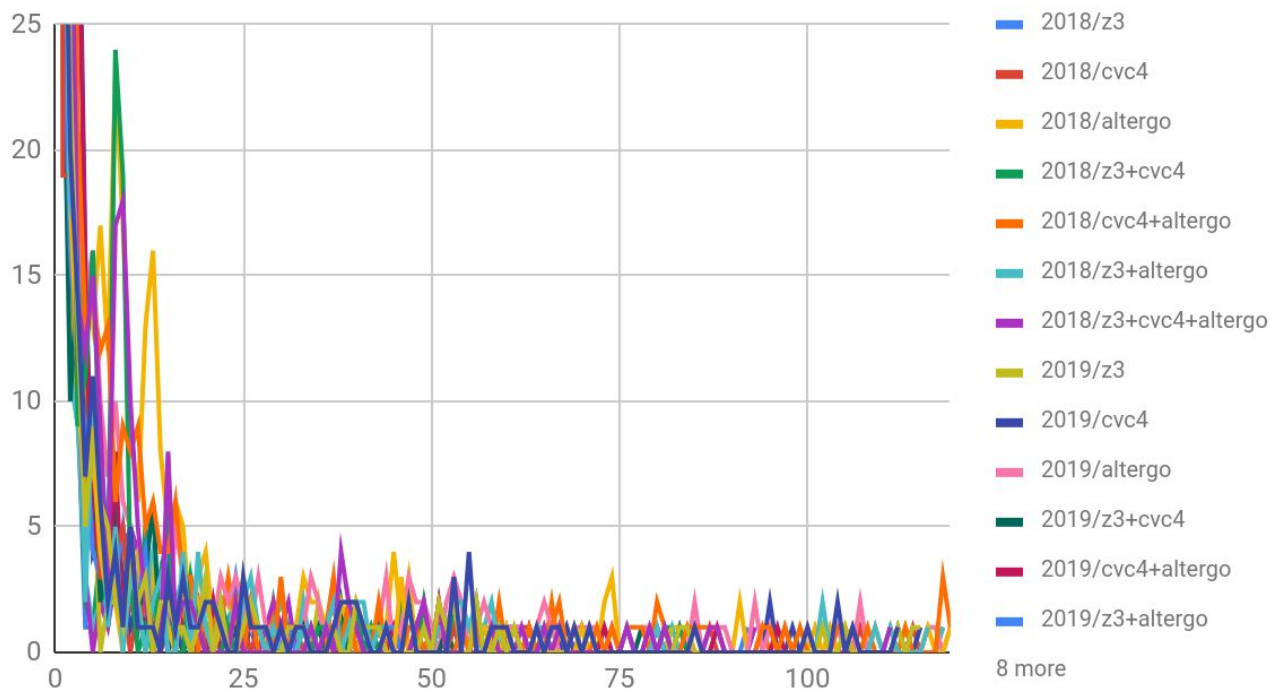
SPARK Red-Black Trees - Max Time - All Runs

Pushing timeout to 120 seconds gives almost same results

Long queue of checks only proved at higher timeouts

Except for all provers in 2019

number of proved checks by max time



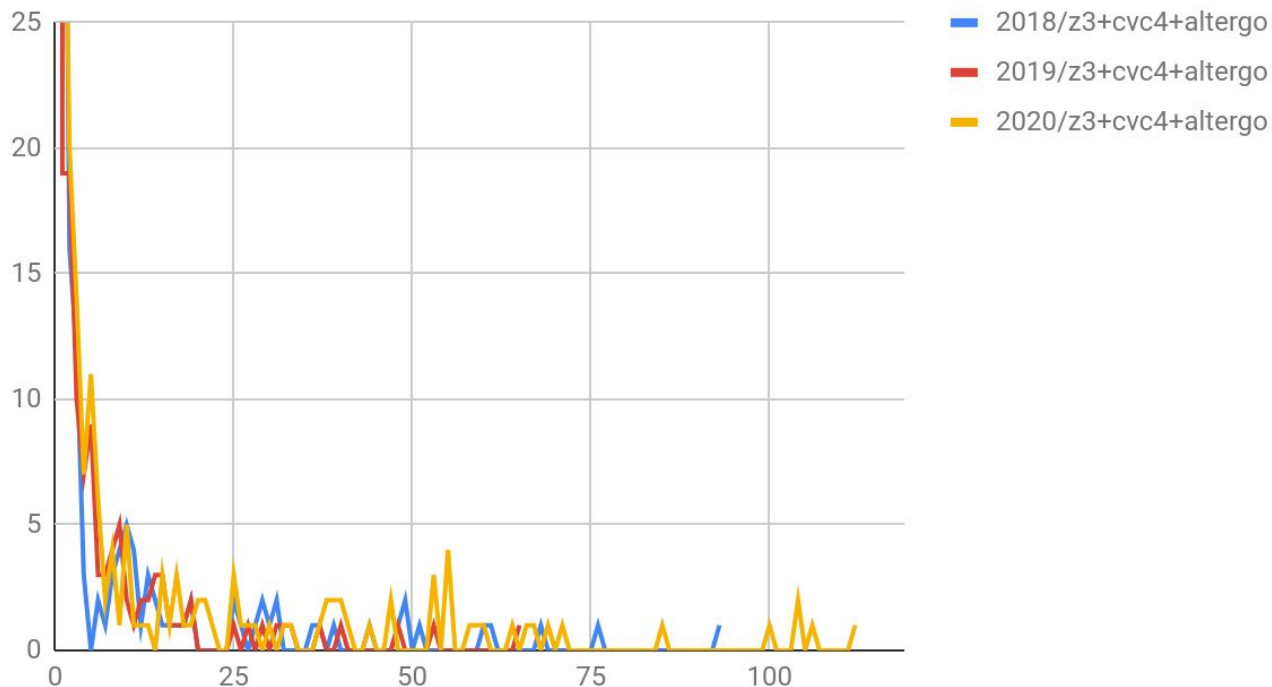
SPARK Red-Black Trees - Max Time - Across Versions

Pushing timeout to 120 seconds gives almost same results

Long queue of checks only proved at higher timeouts

Except for all provers in 2019

number of proved checks by max time - all provers



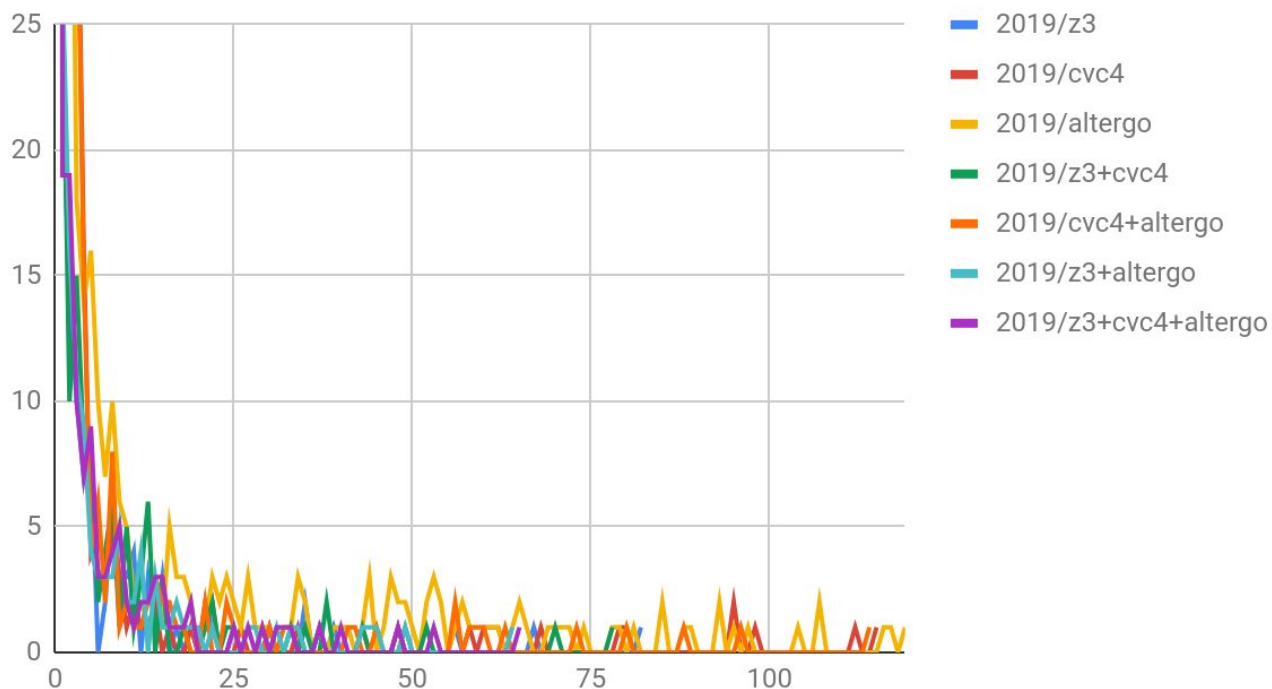
SPARK Red-Black Trees - Max Time - Version 2019

Pushing timeout to 120 seconds gives almost same results

Long queue of checks only proved at higher timeouts

Except for all provers in 2019

number of proved checks by max time - 2019





The Takeaway


Polyprovism or Monoprovism?

GNATprove explicitly exploits polyprovism by diversifying its strategies for VC generation depending on the prover

SPARK projects implicitly exploit polyprovism by not restricting to one prover

In such a setting, polyprovism helps with robustness

Critical role of IVL here as shepherd  or matchmaker 

 Why3: Shepherd Your Herd of Provers, 2011

 Why3 — Where Programs Meet Provers, 2013

To Assert or Not To Assert?

Assertions are just a special case of ghost code

Absolutely required in auto-active proof

But remember `Is_Heap`:

can we write more assertions like:
that increase robustness,
instead of just assertions like:
that are just part of the process?

```
if H.A (Parent) < H.A (Child) then
  pragma Assert (Parent = Heap_Parent (Child));
  return False;
end if;
```

```
pragma Assert (if Child = H.Size then Is_Heap_Def (H));
if Child mod 2 = 1 then
  Parent := Parent + 1;
end if;

end loop;
pragma Assert (Is_Heap_Def (H));
```

For that, we need tool support that does not currently exist!

What Tool Support for Robustness?

```
8  function Is_Heap
9      (H : Heap)
10     return Boolean with
11     Post  $\Rightarrow$  Is_Heap'Result = Is_Heap_Def (H);
12
13 end Is_Heap_P;
```

Messages Locations Python

🔍 filter

- ▼ 📦 Builder results (1 item in 1 file)
 - ▼ 📄 is_heap_p.ads (1 item)

11:15 low: postcondition proved by a single prover (Alt-Ergo)

Polyprovism and Assertionism: Better Together?

Quite common that a check is not proved by A+B provers, but:
 assertion is proved by A
 check is proved by B when assuming assertion

On the other hand, plethora of assertions increase proof context and lead to loss of proofs

Careful use of ghost code isolates assertions in lemmas

More of an art than a science today