mCoq: Mutation Proving for Analysis of Verification Projects

Karl Palmskog

https://proofengineering.org

Joint work with Ahmet Celik, Marinela Parovic, Emilio Jesús Gallego Arias, and Milos Gligoric

Proof Assistants and Large-Scale Software Systems

Project	**Domain**	**Assistant**	**LOC**
CompCert	compiler	Coq	120k+
seL4	kernel	Isabelle/HOL	200k+
BilbyFS	file system	Isabelle/HOL	14k+
Verdi Raft	k/v store	Coq	50k+

Successes of Verified Software

"[T]he under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task."

Yang et al., PLDI '11

"[No] bugs were found in the distributed protocols of verified systems, despite that we specifically searched for protocol bugs and spent more than eight months in this process."

Fonseca et al., EuroSys '17

Problem: Incomplete and Missing Specifications

"This [miscompilation] bug and five others like it were in CompCert's unverified front-end code. Partly in response to these bug reports, the main CompCert developer expanded the verified portion of CompCert."

Yang et al., PLDI '11

"[W]e have found 16 bugs in the verified systems that have a negative impact on the server correctness or on the verification guarantees. [...] analyzing their causes reveals a wide range of mismatched assumptions [...]."

Fonseca et al., EuroSys '17

mit-plv / **chapar**

Watch 13

Code | Issues 3 | Pull requests 0 | Projects 0 | Wiki | Pulse | Graphs

Message duplication and reordering causes violation of causal consistency #3

Open **pfons** opened this issue on 21 Apr 2016 · 1 comment

pfons commented on 21 Apr 2016

The algorithm 2 produces results that violate causal consistency when messages are duplicated and reordered by the network. This situation can occur in practice because the UDP protocol, which is used for the server-server communication, does not guarantee in-order delivery nor does it guarantee at-most once delivery.

The following steps are sufficient to reproduce the bug, with Client A, Client B and Client C running on different servers:

```
Req 1: Client A: PUT key, “NA”
Req 2: Client A: PUT key, “Request”

Req 3: Client B: GET key -> “Request”
Req 4: Client B: PUT key-effect, “Reply”

<replay packets sent by Req 1>

Req 5: Client C: GET key-effect -> “Reply”
Req 6: Client C: GET key -> “NA”
```

The steps above show that client C can see the effect event (“Reply”), produced by Client B, without seeing the cause event (“Request”), produced by Client A.

Response by Author

MohsenLesani commented on 4 Jul 2016

The semantics in the paper models reordering of messages but not message duplication. Duplication can be modularly handled by transformers presented in Verdi paper.
http://verdi.uwplse.org/verdi.pdf

Mutation Testing

1. make small changes resembling faults to software system
2. execute accompanying test suite on changed system
3. measure how well the test suite catches introduced faults
4. improve test suite and repeat

Examples: Major mutation framework, PIT mutation testing

Our Working Analogy: Proofs ∼ Tests

- tests are "partial functional specifications" of programs
- proofs represent many, usually an infinite number of, tests

```
Fixpoint app {A} (l m:list A)
:= match l with
   | [] ⇒ m
   | a :: l' ⇒ a :: app l' m
   end.
```

1. Coq function

```
Lemma asoc: ∀ A (l m n:list A),
app l(app m n) = app(app l m) n.
Proof.
induction l; intros; auto.
simpl; rewrite IHl; auto.
Qed.
```

2. Coq lemma

```
let test_app_assoc ctxt =
 assert_equal
  (app [1] (app [2] [3]))
  (app (app [1] [2]) [3])
```

3. OCaml test

Our Contributions

1. propose **mutation proving** for deductive program verification
2. implement mutation proving in Coq tool, mCoq
3. evaluate mCoq on 12 large and medium scale Coq projects

Mutation Proving

- a **mutation operator** *op* is applied to a Coq project
- *op* may generate a **mutant** where specifications are different
- an *op* mutant where a proof fails during checking is **killed**
- a *op* mutant where all proofs are successfully checked is **live**

Mutation Operators

Category	Name	Description
General	GIB	Reorder branches in if-else expression
	GIC	Reverse constructor order in inductive type
	GME	Replace exp in the 2nd match case with 1st case exp
Lists	LRH	Replace list with head singleton list
	LRT	Replace list with its tail
	LRE	Replace list with empty list
	LAR	Reorder arguments to the list append operator
	LAF	Replace list append expression with first argument
	LAS	Replace list append expression with second argument
Numbers	NPM	Replace plus with minus
	NZO	Replace zero with one
	NSZ	Replace successor constructor with zero
	NSA	Replace successor constructor with its argument
Booleans	BFT	Replace `false` with `true`
	BTF	Replace `true` with `false`

Example Mutation Using GIB

```
Require Import Arith.

Definition update {A} (st : nat → A) h (v : A) :=
 fun n ⇒ if Nat.eq_dec n h then v else st n.

Lemma update_nop : ∀ A (st : nat → A) y v,
 st y = v → update st y v y = st y.
Proof.
 intros; unfold update; case Nat.eq_dec; auto.
Qed.

Lemma update_diff : ∀ A (st : nat → A) x v y,
 x ≠ y → update st x v y = st y.
Proof.
 intros; unfold update.
 case Nat.eq_dec; congruence.
Qed.
```

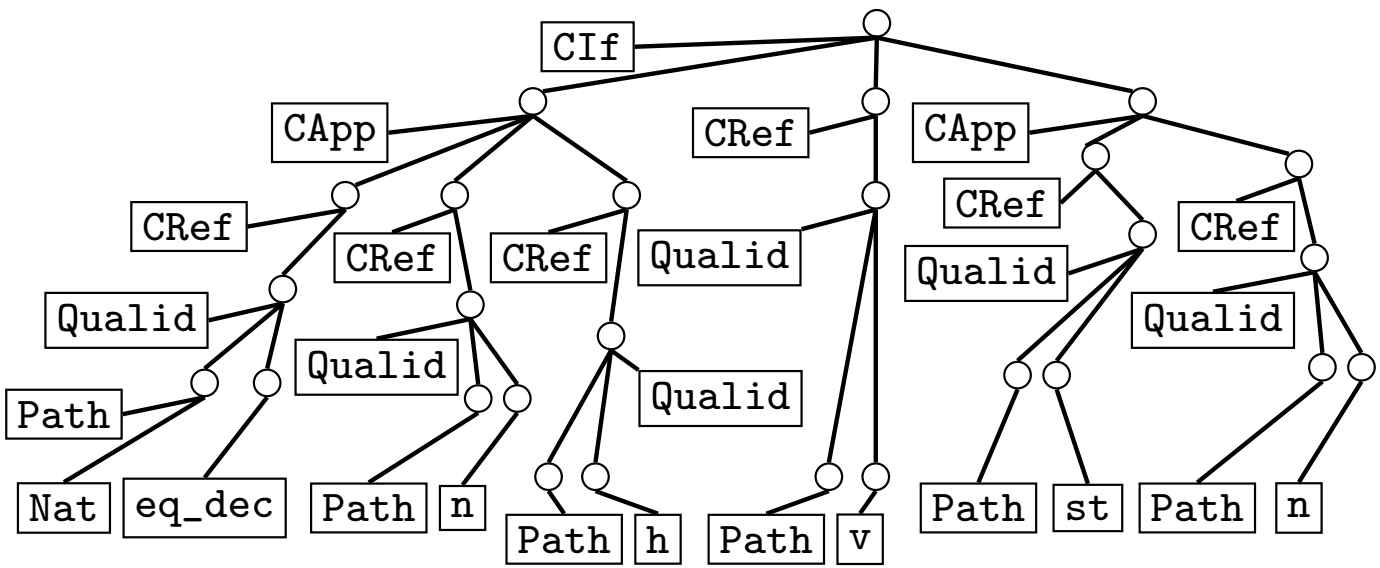
Example, Mutated

```
Require Import Arith.

Definition update {A} (st : nat → A) h (v : A) :=
 fun n ⇒ if Nat.eq_dec n h then st n else v.

Lemma update_nop : ∀ A (st : nat → A) y v,
 st y = v → update st y v y = st y.
Proof.
 intros; unfold update; case Nat.eq_dec; auto.
Qed.

Lemma update_diff : ∀ A (st : nat → A) x v y,
 x ≠ y → update st x v y = st y.
Proof.
 intros; unfold update.
 case Nat.eq_dec; congruence.
Qed.
```

Implementation Approach: S-expression Serialization

```
if Nat.eq_dec n h then st n else v
```

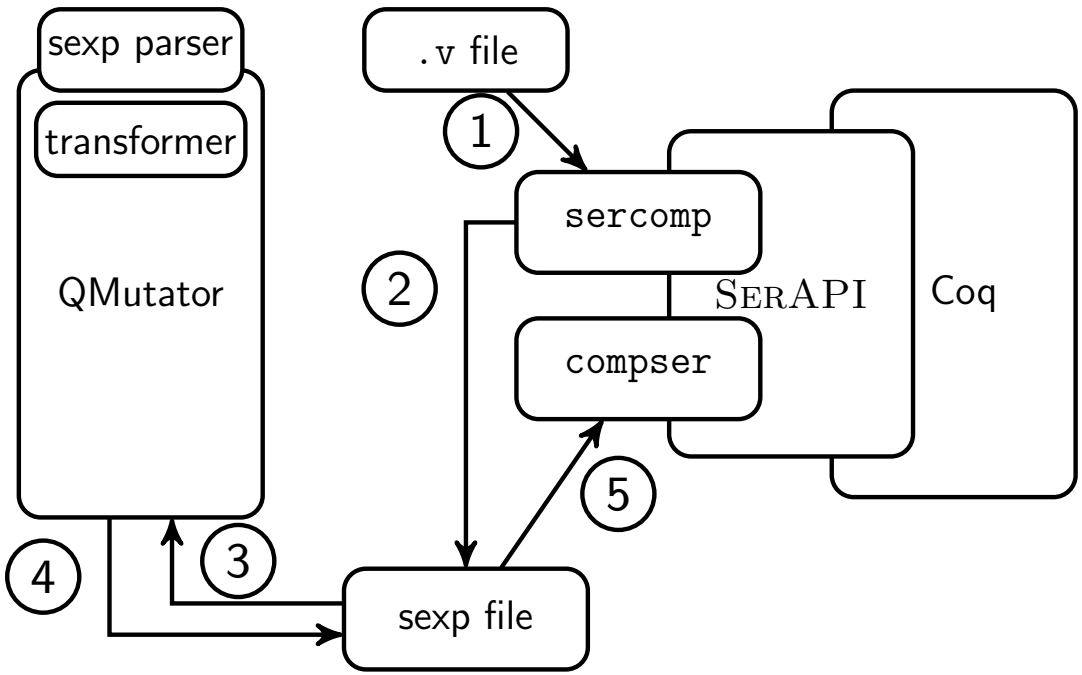
```
(CIf(CApp(()(CRef(Qualid(Path((Id Nat)))(Id eq_dec)) ...))))
```



mCoq Components

`sercomp`	command-line SerAPI-based OCaml program which takes Coq `.v` file and outputs lists of sexps
`compser`	command-line SerAPI-based program which takes lists of sexps and outputs `.vo` file or checks all sexps
Coq fork	fork of the `v8.9` branch of Coq on GitHub to expose key datatypes to SerAPI
SerAPI	extended OCaml library to support full (de)serialization of Coq code, including tactics
QMutator	sexp transformation library in Java that performs operator mutations
Runner	driver program in Java and bash to orchestrate components and compute mutation scores

mCoq Architecture and Workflow



Optimizations: mCoq Modes

Default	simple mode which compiles every file in topological dependency order.
RDeps	advanced mode which checks only affected files and caches and reverts `.vo` files.
Skip	advanced mode which checks only affected files, and also avoids reverting `.vo` files
Noleaves	like Default, but avoids writing leaf files to disk.
ParFile	Like Skip, but parallelizes checking of **files**.
ParQuick	Like Skip, but parallelizes checking of **proofs**.
ParMutant	Like RDeps, but checks each mutant in parallel.
6-RDeps	Organizes operators into six groups, and runs each group in parallel using RDeps.

Procedure

Require: G – Dependency Graph
Require: rG – Reverse Dependency Graph
Require: op – Mutation operator
Require: $sVFs$ – Topologically sorted `.v` files
Require: v – Set of visited `.v` files
Require: vF – `.v` file

1: **procedure** CHECKOPVFILE(G, rG, op, $sVFs$, v, vF)
2: $sF \leftarrow$ `sercomp`(vF)
3: $mc \leftarrow countMutationLocations(sF, op)$
4: $mi \leftarrow 0$
5: **while** $mi < mc$ **do**
6: $mSF \leftarrow mutate(sF, op, mi)$
7: CHECKOPSEXPFILE($G, rG, sVFs, v, vF, mSF$)
8: $mi \leftarrow mi + 1$
9: **end while**
10: **revertFile**(vF)

Evaluation Research Questions

RQ1 What is the number of mutants of projects and what are their mutation scores?

RQ2 What is the cost of mutation proving in terms of execution time and what are benefits of optimizations?

RQ3 Why are some mutants (not) killed?

RQ4 How does mutation proving compare to dependency analysis for finding incomplete and missing specifications?

Evaluation: Open Source Git-Based Projects

Project	**#Files**	**Spec. LOC**	**Pr. LOC**
ATBR	42	4123	5567
FCSL PCM	12	2939	2851
Flocq	29	5955	18044
Huffman	26	1878	4011
MathComp	89	37520	46040
PrettyParsing	14	1221	705
Bin. Rat. Numbers	37	5500	29541
Quicksort Compl.	36	2617	6202
Stalmarck	38	3552	7698
Coq-std++	43	6882	6852
StructTact	19	2008	2333
TLC	49	13217	7802
Avg.	36.16	7284.33	11470.50
Total	434	87412	137646

Evaluation Environment

6-core Intel Core i7-8700 CPU @ 3.20GHz machine with 64GB of RAM, running Ubuntu 18.04.1 LTS.

Limit the number of parallel processes to be at or below the number of physical CPU cores.

RQ1: Number of Mutants

Project	**Total**	**Killed**
ATBR	355	335
FCSL PCM	115	112
Flocq	382	349
Huffman	369	366
MathComp	1037	1025
PrettyParsing	282	235
Bin. Rat. Numbers	365	352
Quicksort Compl.	681	637
Stalmarck	565	526
Coq-std++	564	515
StructTact	104	100
TLC	400	306
Avg.	434.91	404.83
Total	5219	4858

RQ1: Mutation Scores

Project	**Score**
ATBR	95.44
FCSL PCM	99.11
Flocq	93.31
Huffman	99.18
MathComp	98.84
PrettyParsing	83.33
Bin. Rat. Numbers	97.23
Quicksort Compl.	93.81
Stalmarck	93.26
Coq-std++	91.63
StructTact	96.15
TLC	76.88
Avg.	93.18

RQ2: Mutation Cost

Project	Checking	Sercomp	Default	RDeps	Skip	Noleaves	ParFile	ParQuick	ParMutant	6-RDeps
ATBR	45.39	131.33	2157.68	1760.27	1761.59	2155.00	1342.52	1523.21	596.21	755.40
FCSL PCM	11.75	21.95	153.22	150.88	151.12	153.47	152.02	150.79	53.33	109.51
Flocq	17.25	37.38	725.82	547.06	547.47	726.71	544.10	543.79	156.63	199.02
Huffman	7.75	11.58	188.64	185.70	186.19	188.13	181.66	207.94	62.46	72.38
MathComp	341.33	593.19	9962.99	8480.79	8482.90	9967.52	6886.28	6763.25	4053.67	3943.05
PrettyParsing	4.37	5.57	278.56	216.98	217.24	278.67	214.50	268.35	66.06	90.21
Bin. Rat. Numbers	26.29	16.95	1022.61	925.50	925.80	1022.19	894.52	889.60	264.85	578.94
Quicksort Compl.	17.66	34.33	1594.66	1064.64	1062.81	1596.87	914.65	928.41	362.38	553.53
Stalmarck	9.21	16.55	805.84	498.01	499.00	803.52	469.42	571.76	192.78	230.62
Coq-std++	30.94	57.01	3187.80	2597.54	2597.34	3186.81	2194.68	2403.13	776.77	1137.16
StructTact	3.40	7.27	55.90	41.62	40.98	55.93	39.72	40.20	18.84	19.35
TLC	21.82	44.77	3128.85	1739.27	1738.99	3126.18	1467.15	1542.01	519.59	693.88
Avg.	44.76	81.49	1938.54	1517.35	1517.61	1938.41	1275.10	1319.37	593.63	698.58
Total	537.16	977.88	23262.57	18208.26	18211.43	23261.00	15301.22	15832.44	7123.57	8383.05

RQ3: Why are some mutants (not) killed?

We manually inspected 74 live mutants (out of 361), which we labeled with one of:

- UnderspecifiedDef: The live mutant pinpoints a definition which lacks lemmas for certain cases (33 mutants).
- DanglingDef: The live mutant pinpoints a definition that has no associated lemma (30 mutants).
- SemanticallyEq: The live mutant is semantically equivalent to the original project (11 mutants).

RQ3: MathComp Live LRT Mutant

```
Fixpoint merge_sort_push s1 ss :=
match ss with
| [::]   ::  ss' | [::]   as ss' ⇒ s1 :: ss'
| s2 ::  ss' ⇒
  [::]   ::  merge_sort_push (merge s1 s2) ss'
end.
```

RQ3: MathComp Live LRT Mutant, Mutated

```
Fixpoint merge_sort_push s1 ss :=
match ss with
| [::]   ::  ss' | [::]   as ss' ⇒ s1 :: ss'
| s2 ::  ss' ⇒
  merge_sort_push (merge s1 s2) ss'
end.
```

RQ3: MathComp Live LRT Mutant, Commented

```
Fixpoint merge_sort_push s1 ss :=
match ss with
| [::]   ::  ss' | [::]   as ss' ⇒ s1 :: ss'
| s2 ::  ss' ⇒
  merge_sort_push (merge s1 s2) ss'
end.
```

> [T]he key but unstated invariant of `ss` is that its ith item has size 2^i if it is not empty, so that `merge_sort_push` only performs perfectly balanced merges [...] without the `[::]` placeholder the MathComp sort becomes two element-wise insertion sort.
>
> —Georges Gonthier

RQ3: Flocq Live GIB Mutant

```
Definition Bplus op_nan m x y :=
match x,y with
| B754_infinity sx, B754_infinity sy ⇒
  if Bool.eqb sx sy then x
  else build_nan (plus_nan x y)
```

RQ3: Flocq Live GIB Mutant, Mutated

```
Definition Bplus op_nan m x y :=
match x,y with
| B754_infinity sx, B754_infinity sy ⇒
  if Bool.eqb sx sy then build_nan (plus_nan x y)
  else x
```

RQ3: Flocq Live GIB Mutant, Commented

```
Definition Bplus op_nan m x y :=
match x,y with
| B754_infinity sx, B754_infinity sy ⇒
  if Bool.eqb sx sy then build_nan (plus_nan x y)
  else x
```

- Bplus lemmas rule out infinite cases through guards
- same problem with Bminus function
- more lemmas may be needed

RQ4: Comparison to dependency analysis

- compared to grep-based baseline ("do names occur in source files?")
- compared to term dependency extraction ("do names occur in elaborated terms?")
- conclusion: baseline is useless, term dependency lists are noisy

See paper for details!

Conclusion

- technique for analyzing proof assistant projects
- Coq tool, mCoq, implementing technique and optimizations
- evaluation shows mCoq finds incomplete/missing specs
- paper accepted to ASE, link will appear on `https://proofengineering.org`

Contact us:

- Ahmet Celik, `ahmetcelik@utexas.edu`
- **Karl Palmskog**, `palmskog@acm.org`
- Marinela Parovic, `marinelaparovic@gmail.com`
- Emilio Jésus Gallego Arias, `e@x80.org`
- Milos Gligoric, `gligoric@utexas.edu`

MathComp Merge Sort

```
Fixpoint merge_sort_push s1 ss :=
  match ss with
  | [::]   ::  ss' | [::]   as ss' ⇒ s1 :: ss'
  | s2 ::  ss' ⇒ [::]  ::  merge_sort_push (merge s1 s2) ss'
  end.

Fixpoint merge_sort_pop s1 ss :=
  if ss is s2 ::  ss' then merge_sort_pop (merge s1 s2) ss' else s1.

Fixpoint merge_sort_rec ss s :=
  if s is [::  x1, x2 & s'] then
    let s1 := if leT x1 x2 then [:: x1; x2] else [::  x2; x1] in
    merge_sort_rec (merge_sort_push s1 ss) s'
  else merge_sort_pop s ss.

Definition sort := merge_sort_rec [::].
```