Mutation Analysis for Coq

Ahmet Celik*, Karl Palmskog*, Marinela Parovic*, Emilio Jesús Gallego Arias† and Milos Gligoric*
*The University of Texas at Austin †MINES ParisTech
ahmetcelik@utexas.edu, palmskog@acm.org, marinelaparovic@gmail.com, e@x80.org, gligoric@utexas.edu

***Abstract*—Mutation analysis, which introduces artificial defects into software systems, is the basis of mutation testing, a technique widely applied to evaluate and enhance the quality of test suites. However, despite the deep analogy between tests and formal proofs, mutation analysis has seldom been considered in the context of deductive verification. We propose mutation proving, a technique for analyzing verification projects that use proof assistants. We implemented our technique for the Coq proof assistant in a tool dubbed MCOQ. MCOQ applies a set of mutation operators to Coq definitions of functions and datatypes, inspired by operators previously proposed for functional programming languages. MCOQ then checks proofs of lemmas affected by operator application. To make our technique feasible in practice, we implemented several optimizations in MCOQ such as parallel proof checking. We applied MCOQ to several medium and large scale Coq projects, and recorded whether proofs passed or failed when applying different mutation operators. We then qualitatively analyzed the mutants, finding many instances of incomplete specifications. For our evaluation, we made several improvements to serialization of Coq files and even discovered a notable bug in Coq itself, all acknowledged by developers. We believe MCOQ can be useful both to proof engineers for improving the quality of their verification projects and to researchers for evaluating proof engineering techniques.**

I. Introduction

Mutation analysis introduces small-scale modifications to a software system, with each modified system version called a *mutant*. Mutation analysis is widely applied to software systems to perform *mutation testing* [1], where test suites are evaluated on mutants of a system that represent faults introduced by programmers, or are designed to give rise to fault-like behavior. If a specific mutant induces test failures, the mutant is said to be *killed*; otherwise it is said to be *live*. However, if a mutant survives all tests, this may indicate an inadequate test suite or present avenues to improve tests. Mutants of a system can be produced in a variety of ways; a common approach implemented for many programming languages, including functional languages such as Haskell, is to apply *mutation operators* at a level near the source code syntax, e.g., changing $+$ to $-$. An operator may intuitively represent a particular flaw that programmers are prone to make, such as getting the sign of an integer variable wrong.

Formal verification can offer guarantees about program behavior and other properties beyond those of testing. In particular, deductive verification using proof assistants is increasingly used for development of trustworthy large-scale software systems [2]–[5]. Nevertheless, just as test suites may be inadequate, formal specifications can fail to account for unwanted program behavior [6], [7], potentially compromising the ability of formal verification to rule out bugs and leading to lower trust in verified code.

Although it is regularly applied to unverified software and during lightweight verification [8], [9], mutation analysis has only rarely been considered for proof assistants [10], and to our knowledge, never with formal proofs in place of tests.

We propose *mutation proving*, a technique for mutation analysis of verification projects using proof assistants, suitable for evaluating the adequacy of collections of formally proven properties of programs. Our technique adapts and extends mutation operators previously used to mutate Haskell programs [11], [12]. We implemented our technique for the Coq proof assistant [13] in a tool dubbed MCOQ. Given a mutation operator and a Coq project, MCOQ applies an instance of the operator to a definition in Coq's Gallina language, and then checks all proofs that could be affected by the change.

A serious obstacle to operator-based mutation analysis in proof assistants is the extensibility and flexibility of the syntax used to express functions, datatypes, and properties. In particular, Coq supports defining powerful custom *notations* over existing specifications [14], and Coq's parser can be extended with large grammars at any point in a source file by loading plugins [15]. These facilities are convenient for expressing mathematical concepts, but pose a great challenge for processing of Coq files. Moreover, definitions of functions and datatypes, analogous to classes and methods in Java-like languages, tend to be highly interspersed with proofs, which are analogous to tests [16]. This precludes simple mutation based on text replacement in source files [17].

We overcome these challenges by leveraging the OCaml-based SERAPI serialization library [18], which is integrated with Coq's parser and internal data structures. We extended Coq and SERAPI to support full serialization of all Coq files used in large-scale projects to *S-expressions* (sexps) [19]. We apply our mutation operators to the sexps we obtain, and then deserialize and proof-check the results. To make mutation proving feasible in practice for large-scale Coq projects, we optimized MCOQ in several ways, e.g., to leverage multi-core hardware for fast parallel checking of proofs affected by changes after applying a mutation operator.

To evaluate our technique, we applied MCOQ to several open source Coq projects, from medium to large scale. We recorded whether a mutant was live or killed based on proofs passing or failing, and then qualitatively analyzed a subset of mutants, unveiling several incomplete specifications. For our evaluation, we enhanced SERAPI and fixed several serialization issues, significantly increasing its robustness in processing large Coq projects. We also found a notable bug in Coq related to proof processing when applying MCOQ [20], acknowledged and subsequently fixed by the developers [21].

Our technique and tool can be useful both to proof engineers for directly analyzing their verification projects and to researchers for evaluating proof engineering techniques, analogously to how mutation testing is used to evaluate testing techniques for functional programs [12].

We believe mutation proving is largely orthogonal to, and complements, many other analysis techniques for proof assistants, such as bounded testing [22], dependency analysis [16], [23], counter-example generation [24], [25], property-based testing [26], [27], and theory exploration [28]. Specifically, these techniques do not consider "alternative worlds", where definitions are different from the present ones [8].

We make the following contributions:

- **Technique**: We propose *mutation proving* for verification projects using proof assistants. We define a set of mutation operators on definitions of functions and datatypes, inspired by operators defined previously for functional and imperative programming languages.
- **Tool**: We implemented mutation proving in a tool, dubbed MCOQ, which supports Coq projects. Our tool brings significant extensions to Coq and the SERAPI library for serialization and deserialization of Coq syntax; these extensions pave the way for other transformations of Coq code.
- **Optimizations**: To make mutation proving of large projects feasible in practice, we optimized MCOQ to make it run faster. In particular, we implemented several novel forms of selective and parallel checking of proofs for mutants.
- **Evaluation**: We performed an empirical study using MCOQ on 12 large and medium-sized open source Coq projects. For each project, we recorded the number of generated and killed mutants and the execution time. We qualitatively analyzed a subset of the mutants, and found several incomplete specifications manifested as live mutants.
- **Impact**: Our work resulted in many improvements and bugfixes to SERAPI, enhancing its robustness when applied to large-scale projects and showing that complex, extensible Coq files can be manipulated in a lightweight way. We made several modifications to Coq itself, and these changes have been accepted by Coq developers.

We provide supplementary material and artifacts related to MCOQ at: http://cozy.ece.utexas.edu/mcoq

II. Background

This section provides some brief background on the Coq proof assistant, the SERAPI library, and mutation testing.

A. The Coq Proof Assistant

Coq is a proof assistant based on type theory [13], implemented in the OCaml programming language. The specification language of Coq, Gallina, is a small and purely functional programming language. Proofs about Gallina specifications are typically performed using sequences of expressions (tactic calls) in Coq's proof tactic language, Ltac [29]. Source files processed by Coq are sequences of *vernacular commands*, each of which can contain both Gallina and Ltac expressions. Figure 1 shows an example Coq source file which contains a function `update` and two lemmas about the function. The intended meaning of `update`, defined on lines 3–4, is that it returns a new version of a given function `st` from natural numbers to some type `A`, and this returned function maps `h` to `v` but otherwise behaves as `st`.

```
 1 Require Import Arith.
 2
 3 Definition update {A} (st : nat → A) h (v : A) :=
 4  fun n ⇒ if Nat.eq_dec n h then v else st n.
 5
 6 Lemma update_nop : ∀ A (st : nat → A) y v,
 7  st y = v → update st y v y = st y.
 8 Proof.
 9  intros; unfold update; case Nat.eq_dec; auto.
10 Qed.
11
12 Lemma update_diff : ∀ A (st : nat → A) x v y,
13  x ≠ y → update st x v y = st y.
14 Proof.
15  intros; unfold update.
16  case Nat.eq_dec; congruence.
17 Qed.
```

Update.v

Fig. 1: Example Coq source file.

Vernacular syntax is extensible by the user in almost arbitrary ways by (1) defining *notations* inside Coq, e.g., `[]` or `[::]` for the empty list constructor `nil`, and (2) loading plugins in Coq that extend syntax. In particular, the Ltac language and basic decision procedures for proof automation are implemented as a collection of plugins. Since plugins can generally be loaded at any time when interacting with Coq, the permitted syntax can grow dynamically as a vernacular file is processed. Hence, writing a robust stand-alone parser for vernacular is difficult, and will break easily as Coq evolves.

Even though Coq provides a logic of total, terminating functions, Ltac allows nontermination, e.g., of proof search. Hence, modifying a Gallina datatype or function may result in infinite loops, in analogy with the frequent infinite loops that arise in tests during mutation testing [1]. The mitigating practice in mutation testing is to assign execution time thresholds for test execution. Similarly, we set thresholds to the proof checking time for each mutant.

The `coqc` tool compiles source `.v` files to binary `.vo` files and checks all proofs. Such binary files are then loaded by Coq when processing `Require` commands in `.v` files.

B. SERAPI *and Serialization to S-expressions*

SERAPI is an OCaml library and toolchain for machine interaction with Coq [18]. SERAPI has two principal components: (1) an interface for serialization and deserialization of Coq syntax and internal data structures to and from S-expressions (sexps) [19] built on OCaml's PPX metaprogramming facilities [30], and (2) a protocol for building and querying Coq files that abstracts over vernacular commands. In effect, SERAPI overcomes the problem of robustly parsing vernacular by directly integrating with Coq's parsing toolchain

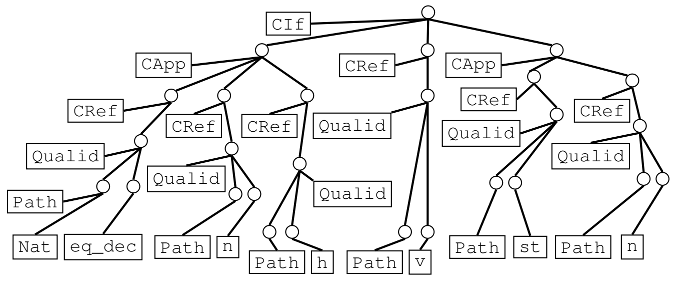

Fig. 2: Sexp of if-subexpression on line 4 in Figure 1.

and internal datatypes. Since the serialization routines are automatically generated from Coq's own definitions using metaprogramming, SERAPI is expected to require only modest maintenance as Coq evolves. Before our work, the principal application of SERAPI was for user interfaces for Coq, e.g., web-based interfaces [31].

When mutating Coq projects, we use the SERAPI sexp-based serialization facilities, avoiding heavyweight OCaml library development. Intuitively, a SERAPI sexp is either an atom, representing a constant or variable name, or a list delimited by parentheses. For example, the sexp for the command on the first line in Figure 1 is as follows:

```
(VernacExpr()(VernacRequire()
 (false)(((Qualid(Path)(Arith))))))
```

A more readable but less compact representation of sexps is graphically as trees. For example, the tree in Figure 2 provides a simplified illustration of the SERAPI sexp for the if-subexpression on line 4 in Figure 1.

C. Mutation Testing and Proving

We follow Papadakis et al. [1] in using *mutation analysis* for the process of generating code variants, and *mutation testing* for the application of this process to support software testing and test suite improvement. In analogy with the latter, we refer to the application of mutation analysis to support proof development using proof assistants and improving collections of formally proven properties as *mutation proving*.

Mutation analysis was proposed by Lipton, then formalized by DeMillo et al. [32], and first applied in practice in the context of software testing by Budd et al. [33]. In mutation testing, test suites that distinguish between a mutant and the original program, e.g., by reporting an assertion violation, leaving the mutant *killed*, are judged to meet objectives. In contrast, test suites that do not report assertion violations or other errors for a mutant, leaving the mutant *live*, could be judged not to meet objectives and may require revision. A test suite's *mutation score* is defined as the percentage of killed mutants out of all mutants that are distinct under *functional equivalence* [34]. Intuitively, mutants may be viewed as containing buggy code, and the mutation score as a measure of how well the test suite rules out the presence of buggy code [35], [36].

How to interpret killed and live mutants in mutation proving is less clear than for mutation testing. While there may be definitions of functions and datatypes that are nonsensical for most purposes, a failing proof of a lemma using such definitions does not unambiguously indicate an error or mistake (bug) in the definitions. Coq proof scripts are often brittle [37] and fail to produce proofs when associated definitions are changed in trivial ways that preserve all their properties. In addition, the goal of a proof assistant verification project may be to prove some lemma unrelated to any specific program.

Nevertheless, live mutants may still indicate the inadequacy of the verification *harness* [8] to fully meet reasonable objectives. In particular, live mutants can go *far beyond* flagging up completely unused definitions as in dependency analysis [23]: they can pinpoint that certain *fragments* of key definitions vacuously satisfy behavioral specifications [9], e.g., that an ostensibly strong and complete lemma about a function can be proven regardless of what the returned value is for a certain range of inputs to that function. Many live mutants could indicate the presence of such underspecification in a Coq project, which may eventually manifest as bugs in executable systems [6] and lead to lower trust in formally verified code.

III. TECHNIQUE

In this section, we describe our mutation approach, mutation operators, and optimizations to mutation proving.

A. Mutation Approach

Our approach to mutation proving follows the classical approach of defining a set of *mutation operators* (operator for short) which describe classes of changes to a project. Intuitively, an operator captures a common mistake made by a proof engineer. When an operator is successfully applied to a project, it generates a mutant. When the mutant has been successfully checked, i.e., all related proofs have passed, it is declared *live*. Otherwise, if proof checking fails or times out, the mutant is considered *killed*. Since the notion of functional equivalence is not applicable to many verification projects, we use a broader definition of *mutation score* as the percentage of killed mutants out of all mutants that are distinct under *syntactical equivalence*.

We define operators for mutation proving as transformations on Coq vernacular sexps. For any transformation and sexp, it must be unambiguous and easily checkable whether the transformation can be successfully applied or not to the sexp. For example, if the transformation pertains to particular Coq constants, it is applicable precisely when those constants occur in a specific way in the sexp. Note that checking syntactical equivalence of a target and result sexp (modulo non-essential auxiliary data such as file line numbers) is simple and fast.

The initial step for applying any operator to a Coq verification project is to convert all `.v` source files to lists of sexps. For a specific operator *op* and sexp list, the steps are then to (1) apply *op* sequentially to all list elements until a non-equivalent mutant is generated, (2) check the mutated list of sexps, (3) check all proofs in files that transitively depend on the source file that was (indirectly) mutated. The latter three steps are repeated for all lists of sexps until no additional mutants can be generated using *op*.

TABLE I: List of Mutation Operators.

Category	Name	Description
General	GIB	Reorder branches in if-else expression
	GIC	Reverse the order of the constructors in the definition of an inductive type
	GME	Replace expression in the second match case with the expression from the first match case
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`

```
 1 Require Import List. Import ListNotations.
 2
 3 Fixpoint ftmap {A B} (f:A → option B) l:list B :=
 4 match l with
 5 | [] ⇒ [] | a :: xs ⇒
 6   match f a with
 7    | None ⇒ ftmap f xs
 8    | Some b ⇒ b :: ftmap f xs
 9   end
10 end.
11
12 Lemma ftmap_app : ∀ A B (f: A → option B) xs ys,
13  ftmap f (xs ++ ys) = ftmap f xs ++ ftmap f ys.
14 Proof.
15  induction xs; intros; simpl in *; auto.
16  case (f a) eqn:?; simpl; auto using f_equal.
17 Qed.
18
19 Lemma ftmap_in : ∀ A B (f: A → option B) a b xs,
20  f a = Some b → In a xs → In b (ftmap f xs).
21 Proof.
22  induction xs; simpl; auto.
23  case (f a0) eqn:?; simpl; intuition congruence.
24 Qed.
```

Ftmap.v

Fig. 3: Example Coq source file using lists.

B. Mutation Operators

Our inspiration for Coq mutation operators came from two sources. Primarily, we were inspired by the operators defined by Le et al. for Haskell [11]. Secondarily, we took inspiration from the operators in mutation frameworks for Java such as PIT [38] and the Major framework [39], [40]. We considered these operators through the lens of our experience from using Coq for over 17 years (cumulative).

Table I lists our operators. For each operator, we give a category, a short name which we will use in the rest of text, and a short description. The General category includes operators which are applicable regardless of whether a project uses a specific datatype from the Coq standard library. The Lists category includes operators which pertain to the ubiquitous list datatype in the standard library. The Numbers category includes operators which apply to natural numbers in their standard linear-size Peano encoding (e.g., `2` is defined as the successor constructor applied two times to the zero constructor). Similarly, the Booleans category applies to booleans as defined in the standard library.

In contrast to imperative languages such as Java, where numeric datatypes are typically built-in, Gallina has only a few native constructs, which is reflected in the limited number of operators in the General category. Other operators require a project to use the corresponding notations and constants from the standard library; the associated categories therefore pertain to the most elementary and widely used parts of the library.

To illustrate how our operators work, we give a few examples using the Coq code in Figure 1 and Figure 3. For a more intuitive presentation, we describe the effect of operators mostly in terms of the source code rather than sexps.

General mutation example. Applying the operator GIB to the file in Figure 1 results in one mutant where `update` has the expressions `v` and `st n` swapped on line 4. The proof of `update_nop` goes through for the mutant, indicating that the lemma does not express any fundamental property of `update`. However, the proof of `update_diff` fails (specifically, `congruence` on line 16 fails), killing the mutant. Note that the mutation can be performed at the sexp level by swapping the two rightmost subtrees below `CIf` in Figure 2.

Lists mutation examples. The source file in Figure 3 contains a recursive function `ftmap` (lines 3–10) that applies a given partial function `f` to a list. The two accompanying lemmas express some basic properties about the function; in particular, `ftmap_app` establishes that `ftmap` distributes over list append. Applying the operator LRH results in a mutant where the singleton list `[b]` has replaced `b :: ftmap f xs` on line 8. This mutant is killed by `ftmap_app`, since this property no longer holds. Applying the operator LRT results in a mutant where the (tail) list expression `ftmap f xs` has replaced `b :: ftmap f xs` on line 8. This mutant survives `ftmap_app`, but is killed by `ftmap_in`.

C. Mutation Optimizations

Mutation analysis is generally acknowledged to be a costly process [41], [42], and this also holds true for mutation proving. In this section, we describe several optimizations to our basic mutation proving approach.

In mutation testing, optimizations are generally about generating faster, smarter, or fewer mutants [43], [44]. We focus on accomplishing *faster* mutation, and the insight we build on is that proof checking for mutation proving can be viewed as a particular instance of *regression proving*, i.e., to check an existing Coq project after a change has been made; similar insights are found in regression testing [45].

Proof selection. A proof selection technique uses knowledge of *modified* files (or proofs) in a project to only check *impacted* files (or proofs) [16]. Since a successful application of a mutation operator means that a sexp file was modified, we can use change impact analysis to perform selective proof checking during mutation proving.

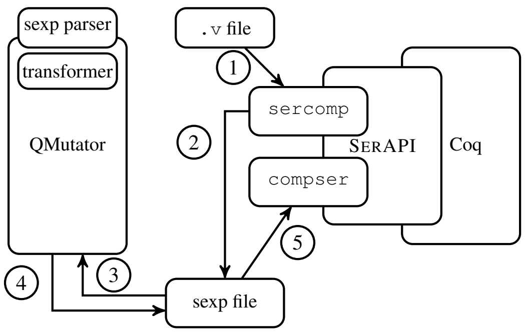

Fig. 4: MCoq implementation architecture.

Proof checking parallelization. Unlike test execution in Java-like languages, proof checking in proof assistants is deterministic, which increases the potential for parallelization on multi-core hardware. In particular, Coq proof checking is routinely parallelized at the file level, where the main restriction on the degree of parallelism is the file dependency graph [46].

Mutation operator parallelization. Since we only perform first-order mutation [47], application of one mutation operator to a project can be performed completely independently of the application of another operator. Hence, when the goal is to apply several operators to the same project, the outcomes can be computed in parallel, as in mutation testing of software [43].

Mutant parallelization. Application of one mutant can be performed completely independently of application of another mutant. We thus also introduce a parallel mode where each mutant is checked as a separate task.

IV. Implementation

In this section, we describe the components of our mutation proving implementation, define and discuss our mutation modes and procedure which use the components, and outline the impact of our tool development on other projects.

A. Tool Architecture and Components

Our tool for mutation proving, dubbed MCoq, is implemented in OCaml, Java, and bash. Figure 4 shows an overview of the architecture of MCoq, and highlights how the main components interact. During mutation proving, Coq source files to be mutated are first given as input to our `sercomp` program integrated with SerAPI (1), which produces corresponding files with lists of sexps (2). The sexps are then handed to our QMutator program (3), which performs parsing and applies the transformations corresponding to a specified mutation operator. Ultimately, QMutator outputs mutated sexps (4) which become input to our `compser` program integrated with SerAPI (5). We next describe each main component of MCoq in detail.

`sercomp`: We implemented a command-line program called `sercomp` on top of SerAPI which takes a regular Coq `.v` source file as input and outputs the corresponding lists of sexps. The program is now included as part of SerAPI [48].

`compser`: We implemented a command-line program called `compser` on top of SerAPI, meant to be the inverse of `sercomp`. `compser` takes a file with a list of sexps as input and either produces a `.vo` file or simply checks every sexp. The program is now included as part of SerAPI [48].

Coq fork: We forked the `v8.9` branch of the Coq GitHub repository corresponding to Coq version 8.9 and modified it to expose internal data structures relevant for mutation proving to SerAPI. We submitted our proposed changes to the Coq repository, and the developers eventually merged them.

SerAPI: We extended SerAPI to provide serialization and deserialization of all Coq internal data structures required to support large projects. In particular, we added support for serialization of Ltac syntax extensions added by the SSReflect proof language [49] used in many projects. All of our changes have been added to the SerAPI codebase.

QMutator: We implemented a library for transformation of sexps produced by `sercomp`, and mutation operators that use this library, in Java. We used an existing library, jsexp [50], to parse and encode sexps. Based on our experience, implementing new operators on top of our library is quick and straightforward. On top of our library, we implemented a program dubbed QMutator that takes sexps and an operator name as input, and produces mutated sexps.

Runner: We implemented a program in Java and bash that uses the above components to perform mutation proving on a given Coq project, and then computes its mutation score.

B. Mutation Modes and Procedure

Based on the approach and optimizations in Section III, we define four basic mutation proving *execution modes*:

Default: A simple mode which checks every file in a project after a mutant is generated, by compiling `.v` files to `.vo` files in topological order according to the file dependency graph.

RDeps: An advanced mode which checks only `.v` files affected by a mutation, and caches and reverts to unmodified `.vo` files to avoid the cost of generating them more than once.

Skip: An advanced mode which checks only `.v` files affected by a mutation, and additionally avoids reverting `.vo` files.

Noleaves: A variant of Default which checks proofs in leaf nodes in the file dependency graph but does not generate `.vo` files for those files. We added this mode to explore if there were any notable speedups gained by avoiding to write `.vo` files with `compser`.

To realize these modes, we implemented the parameterized mutation procedure CheckOp shown in Figure 5 in our Runner program. In the subprocedures called by CheckOp, there are several auxiliary procedures that behave differently depending on the mode:

revertFile: For the Default and Skip modes, the file vF is always reverted. For RDeps, vF is never reverted. For Noleaves, vF is reverted only if it is not a leaf node in rG.

getOtherFiles: For the Default and Noleaves modes, this procedure returns $rG.topologicalSort(sVFs - v)$, whereas for the RDeps and Skip modes, the procedure instead returns $rG.topologicalSort(rG.closure(\{vF\}) - v)$.

Algorithm 1 Pseudocode of CHECKOP.

```
Require: $op$ – Mutation operator
Require: $P$ – Coq Project
1: procedure CHECKOP($op$, $P$)
2:     $vFs \leftarrow P.vFiles()$
3:     $G \leftarrow P.dependencyGraph()$
4:     $rG \leftarrow G.reverse()$
5:     $sVFs \leftarrow rG.topologicalSort(vFs)$
6:     $v \leftarrow \emptyset$
7:     for $vF \in sVFs$ do
8:         $v.add(vF)$
9:         CHECKOPVFILE($G, rG, op, sVFs, v, vF$)
10:    end for
11: end procedure
```

Algorithm 2 Pseudocode of CHECKOPVFILE.

```
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($vF$)
11: end procedure
```

Algorithm 3 Pseudocode of CHECKOPSEXPFILE.

```
Require: $G$ – Dependency Graph
Require: $rG$ – Reverse Dependency Graph
Require: $sVFs$ – Topologically sorted .v files
Require: $v$ – Set of visited .v files
Require: $vF$ – .v file
Require: $mSF$ – Mutated sexp file
1: procedure CHECKOPSEXPFILE($G$, $rG$, $sVFs$, $v$, $vF$, $mSF$)
2:     if compser($mSF$) $\neq 0$ then
3:         $Global.killed[op] \leftarrow Global.killed[op] + 1$
4:         return
5:     end if
6:     $oVFs \leftarrow$ getOtherFiles($G, rG, sVFs, v, vF$)
7:     revertOtherFilesBefore($vF, oVFs$)
8:     for $oF \in oVFs$ do
9:         if coqc($oF$) $\neq 0$ then
10:            $Global.killed[op] \leftarrow Global.killed[op] + 1$
11:            break
12:        end if
13:    end for
14:    revertOtherFilesAfter($vF, oVFs$)
15: end procedure
```

Fig. 5: Mutation procedure pseudocode.

revertOtherFilesBefore: For all modes except Skip, this procedure does nothing. For the Skip mode, it reverts all files in $G.closure(oVFs) - oVFs - \{vF\}$, with $oVFs$ defined on line 6 in CHECKOPSEXPFILE.

revertOtherFilesAfter: For all modes except RDeps, it does nothing. For RDeps, it reverts all files in $oVFs$.

On top of the basic modes, we define four *parallel* modes, which we believed could lead to significant speedups:

ParFile: This mode builds on Skip and parallelizes the for loop in the CHECKOPSEXPFILE procedure (lines 8 to 13). Parallelization is at the coarse-grained file level.

ParQuick: Like ParFile, this mode builds on Skip and parallelizes the for loop in the CHECKOPSEXPFILE procedure (lines 8 to 13). However, parallelization is at the fine-grained level of proofs [46], [51].

ParMutant: This mode builds on RDeps, and checks each mutant in parallel, i.e., we parallelize the while loop in the CHECKOPVFILE procedure (lines 5 to 9).

6-RDeps: In this mode, we organize the operators into groups of six or less, and run groups in parallel using the RDeps mode. We limit to six groups to match the number of cores available in our evaluation machine.

C. Impact of Tool Development

Work on our tool implementation resulted in more than 10 merged code contributions to SERAPI. Specifically, we found over 30 failing test cases that were all fixed. Our enhancements to Coq itself have been merged and are set to be included in the upcoming Coq version 8.10.0 release.

When applying mutation proving to a project (StructTact) during our evaluation, we generated a mutant which we checked with both `coqc` and `compser`; the mutant was killed according to the former but not the latter. The discrepancy was due to a serious bug in Coq related to proof processing [20], acknowledged and subsequently fixed by the developers [21]. This shows that mutation proving development has significantly improved general Coq tooling.

V. EVALUATION

We evaluate MCOQ by answering four research questions:

RQ1: What is the number of mutants created for large and medium sized projects and what are their mutation scores?

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

RQ3: Why are some mutants (not) killed?

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

We run all experiments on a 6-core Intel Core i7-8700 CPU @ 3.20GHz machine with 64GB of RAM, running Ubuntu 18.04.1 LTS. We limit the number of parallel processes to be at or below the number of physical CPU cores. We next describe the studied projects, our independent and dependent variables, and our results.

A. Verification Projects Under Study

Table II lists the Coq projects used in our evaluation; all are publicly available. For each project, we show the project name, the latest SHA at the time of our experiments, number of `.v` files, total lines of code (LOC), specification LOC, and proof script LOC. All LOCs are computed using the `coqwc`

TABLE II: Projects Used in the Evaluation.

Project	SHA	#Files	LOC	Spec. LOC	Pr. LOC
ATBR	366ac237	42	9705	4123	5567
FCSL PCM	b34fce32	12	5747	2939	2851
Flocq	7ec13200	29	24000	5955	18044
Huffman	50687911	26	5889	1878	4011
MathComp	91fa7b57	89	82323	37520	46040
PrettyParsing	189a2625	14	1907	1221	705
Bin. Rat. Numbers	7b9cc06d	37	35041	5500	29541
Quicksort Compl.	0a6eed8b	36	8809	2617	6202
Stalmarck	6932ed8a	38	11266	3552	7698
Coq-std++	005887ee	43	13715	6882	6852
StructTact	82a85b7e	19	4341	2008	2333
TLC	4babc16c	49	23494	13217	7802
Avg.	n/a	36.16	18853.08	7284.33	11470.50
Total	n/a	434	226237	87412	137646

tool, which is bundled with Coq. The last two rows of the table show the average and total values across all projects.

We selected the projects based on (1) compatibility with Coq version 8.9, (2) their size and popularity in terms of, e.g., GitHub stars and usage in other Coq projects, and (3) their inclusion of functions and datatypes that can be mutated.

B. Variables

Independent variables. We manipulate two independent variables in our experiments: operator and execution mode. For the former, we use the 15 operators defined in Table I. For the latter, we use the 8 execution modes described in Section IV-B.

Dependent variables. We compute three dependent variables: mutation score, execution cost, and cost reduction. *Mutation score* provides an estimate of the adequacy of formal specifications; this metric is computed as the percentage of killed mutants out of the total number of mutants minus the number of syntactically equivalent mutants. Mutation score is either computed per mutation operator or for all mutants at once. *Execution cost* is the time needed to perform mutation proving; this metric can also be reported per mutation operator or for all mutants at once. *Cost reduction* is the percentage of time saved using various execution modes compared to the time needed to perform mutation proving using the Default mode.

C. Results

1) RQ1: Number of Mutants and Mutation Score: Table III shows the total number of generated mutants for each pair of project (row) and mutation operator (column). Additionally, the last column shows the total number of mutants per project, and the last two rows show the average and total number of mutants per mutation operator. We can observe that GME generates the most mutants, followed by NZO and NPM. On the other hand, NSZ generates the smallest number of mutants, followed by NSA. This indicates that explicit uses of the natural number successor constructor were few for the projects we used in our evaluation. Table IV shows the number of killed mutants for each pair of project and mutation operator.

Table V shows the mutation score for all pairs of projects and mutation operators; n/a indicates mutation score value that cannot be computed because the number of generated mutants is zero. The last column shows the mutation score for *all* mutants in a given project, which is the metric traditionally reported in mutation testing research; these mutation scores vary from 76.88% (for TLC) to 99.18% (for Huffman).

Recall that mutation scores exclude (syntactically) equivalent mutants. However, including equivalent mutants would affect mutation scores only marginally: all but three projects (ATBR, Flocq, and Bin. Rat. Numbers) had two equivalent mutants or less. The GME operator accounted for all 24 equivalent mutants, which were due to pattern matching cases returning the same expression.

It is important to note that mutation scores are much higher than traditionally seen in mutation testing research. We expected such high scores for several reasons. First, as mentioned in Section II-C, many Coq proof scripts are brittle and fail after only trivial changes are made to specifications. Second, even robust proofs tend to be tightly coupled to functions and datatypes, in effect exploring them symbolically rather than relying only on externally observable properties such as outputs. This is what enables proofs to, e.g., establish properties about all members of infinite sets of datatype instances, which is impossible for traditional unit tests. Two projects are outliers in terms of mutation score (PrettyParsing and TLC) and we come back to this below.

Finally, we analyzed the logs of our runs, which record the reason for each mutant being killed, and found that only 2 mutants were killed due to timeout. These two mutants were generated by GME and LRT.

2) RQ2: Performance: Table VI shows the proof checking and mutation proving time (in seconds) for various execution modes. Specifically, the second column shows time to check the project by running the default build commands (`coqc` via `make`) for each project. The third column shows time to process all files in a project with `sercomp`. Recall that we mutate a file by first obtaining the corresponding sexps via `sercomp`, produce a mutant, and then use `compser` to write a `.vo` file back to disk. Clearly, it would be costly to use both `sercomp` and `compser` to proof check all the files in any given project, so we use this combination only on the file being mutated. The fourth column shows time to perform mutation proving using the Default mode. The remaining columns show execution time for mutation proving for optimized modes.

Due to performing unnecessary proof processing, the Default and Noleaves modes are consistently the slowest, typically by a wide margin. Reasonably, RDeps and Skip give consistent speedups, sometimes substantial, over the basic modes (on average 23% over Default). Nevertheless, some projects such as Huffman show only marginal improvement.

We expected parallel modes to perform better than the advanced sequential modes. However, ParFile and ParQuick were only substantially faster than Skip for some large projects, such as MathComp. This may be due to many mutants being killed quickly before realizing the benefits of parallel checking. For nearly all projects, ParMutant is a clear winner over 6-RDeps and others; its average speedup over Default is 70%.

TABLE III: Total Number of Mutants for each Mutation Operator per Project.

Project	**GIB**	**GIC**	**GME**	**LRH**	**LRT**	**LRE**	**LAR**	**LAF**	**LAS**	**NPM**	**NZO**	**NSZ**	**NSA**	**BFT**	**BTF**	**Total**
ATBR	33	21	74	7	7	7	1	1	1	87	43	19	19	17	18	355
FCSL PCM	0	8	13	8	8	8	0	0	0	2	5	0	0	35	28	115
Flocq	39	14	93	0	0	0	0	0	0	71	54	2	2	45	62	382
Huffman	0	15	45	72	72	72	15	15	15	19	5	5	5	7	7	369
MathComp	0	10	73	58	58	58	12	12	12	114	385	0	0	136	109	1037
PrettyParsing	30	8	68	17	17	17	28	28	28	13	16	3	3	3	3	282
Bin. Rat. Numbers	2	10	52	0	0	0	0	0	0	203	79	4	4	5	6	365
Quicksort Compl.	12	15	77	104	104	104	49	49	49	27	18	30	30	6	7	681
Stalmarck	0	25	129	101	101	101	3	3	3	42	6	1	1	25	24	565
Coq-std++	12	31	149	68	68	68	13	13	13	23	20	22	22	28	14	564
StructTact	7	3	30	9	9	9	2	2	2	12	5	5	5	2	2	104
TLC	4	36	71	38	38	38	5	5	5	23	38	33	33	20	13	400
Avg.	11.58	16.33	72.83	40.16	40.16	40.16	10.66	10.66	10.66	53.00	56.16	10.33	10.33	27.41	24.41	434.91
Total	139	196	874	482	482	482	128	128	128	636	674	124	124	329	293	5219

TABLE IV: Total Number of Killed Mutants for each Mutation Operator per Project.

Project	**GIB**	**GIC**	**GME**	**LRH**	**LRT**	**LRE**	**LAR**	**LAF**	**LAS**	**NPM**	**NZO**	**NSZ**	**NSA**	**BFT**	**BTF**	**Total**
ATBR	32	15	67	7	7	7	1	1	1	84	40	19	19	17	18	335
FCSL PCM	0	8	11	8	8	8	0	0	0	2	5	0	0	34	28	112
Flocq	37	14	77	0	0	0	0	0	0	68	54	2	2	37	58	349
Huffman	0	13	45	72	72	72	15	15	15	19	4	5	5	7	7	366
MathComp	0	8	73	58	56	58	11	12	12	113	381	0	0	135	108	1025
PrettyParsing	24	2	62	15	15	15	25	28	28	9	6	2	2	1	1	235
Bin. Rat. Numbers	2	10	47	0	0	0	0	0	0	199	75	4	4	5	6	352
Quicksort Compl.	11	11	73	96	84	104	49	49	49	26	14	29	29	6	7	637
Stalmarck	0	20	124	101	83	101	2	3	3	42	2	1	1	23	20	526
Coq-std++	11	15	139	63	63	64	12	12	12	23	17	22	22	27	13	515
StructTact	7	2	29	9	9	9	2	2	2	12	5	5	5	1	1	100
TLC	4	16	62	38	31	38	5	5	5	12	22	22	17	18	11	306
Avg.	10.66	11.16	67.41	38.91	35.66	39.66	10.16	10.58	10.58	50.75	52.08	9.25	8.83	25.91	23.16	404.83
Total	128	134	809	467	428	476	122	127	127	609	625	111	106	311	278	4858

3) RQ3: Qualitative Analysis: To qualitatively analyze why mutants are killed or live, we sampled live mutants to inspect manually. To ensure diversity among inspected mutants, we set a requirement of inspecting 10% or more of all live mutants for each operator, and 10% of all live mutants for each project. Due to our familiarity with the project, we also decided to inspect *all* live mutants in MathComp. Initially, we randomly chose mutants to inspect from the set of all live mutants. When we had inspected 5% of total, we finished the remaining MathComp mutants and used the distribution among operators and projects for inspected mutants to sample from underrepresented subsets.

In total, we inspected 74 live mutants, which we labeled with precisely one of the following labels:

- 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).

A detailed description of each live mutant with links to their locations in the original source code repositories can be found in the supplementary material at the MCoq website. Here, we first highlight some notable live mutants labeled with UnderspecifiedDef, and then discuss our general experience from the analysis.

GIB mutant in Flocq: A mutant swapped the branches in the if-else expression of the following function for addition of binary IEEE 754 floating-point numbers:

```
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)
```

The mutant reveals that a particular case of binary addition, namely for numbers representing infinities, is not considered by any lemma. Another live GIB mutant showed the same problem for the analogous definition for subtraction, `Bminus`.

BFT mutant in StructTact: A mutant which changed `false` to `true` in a function named `before_func` on lists highlighted that the function was weakly specified in the library:

```
Fixpoint before_func {A} (f : A → bool) g l :=
match l with | [] ⇒ ⊥ | a :: l' ⇒
f a = true ∨ (g a = false ∧ before_func f g l')
end.
```

Further investigation revealed five general lemmas about `before_func` in Verdi Raft [4]; four of these lemmas kill the mutant. Our changes to factor out all five lemmas to StructTact have been merged in both projects.

TABLE V: Mutation Score for each Mutation Operator per Project.

Project	GIB	GIC	GME	LRH	LRT	LRE	LAR	LAF	LAS	NPM	NZO	NSZ	NSA	BFT	BTF	Total
ATBR	96.96	71.42	95.71	100.00	100.00	100.00	100.00	100.00	100.00	96.55	93.02	100.00	100.00	100.00	100.00	95.44
FCSL PCM	n/a	100.00	100.00	100.00	100.00	100.00	n/a	n/a	n/a	100.00	100.00	n/a	n/a	97.14	100.00	99.11
Flocq	94.87	100.00	90.58	n/a	n/a	n/a	n/a	n/a	n/a	95.77	100.00	100.00	100.00	82.22	93.54	93.31
Huffman	n/a	86.66	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	80.00	100.00	100.00	100.00	100.00	99.18
MathComp	n/a	80.00	100.00	100.00	96.55	100.00	91.66	100.00	100.00	99.12	98.96	n/a	n/a	99.26	99.08	98.84
PrettyParsing	80.00	25.00	91.17	88.23	88.23	88.23	89.28	100.00	100.00	69.23	37.50	66.66	66.66	33.33	33.33	83.33
Bin. Rat. Numbers	100.00	100.00	95.91	n/a	n/a	n/a	n/a	n/a	n/a	98.02	94.93	100.00	100.00	100.00	100.00	97.23
Quicksort Compl.	91.66	73.33	97.33	92.30	80.76	100.00	100.00	100.00	100.00	96.29	77.77	96.66	96.66	100.00	100.00	93.81
Stalmarck	n/a	80.00	96.87	100.00	82.17	100.00	66.66	100.00	100.00	100.00	33.33	100.00	100.00	92.00	83.33	93.26
Coq-std++	91.66	48.38	94.55	92.64	92.64	94.11	92.30	92.30	92.30	100.00	85.00	100.00	100.00	96.42	92.85	91.63
StructTact	100.00	66.66	96.66	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	50.00	50.00	96.15
TLC	100.00	44.44	89.85	100.00	81.57	100.00	100.00	100.00	100.00	52.17	57.89	66.66	51.51	90.00	84.61	76.88
Avg.	94.39	72.99	95.71	97.31	92.19	98.23	93.32	99.14	99.14	92.26	79.86	92.99	91.48	86.69	86.39	93.18

TABLE VI: Proof Checking and Mutation Time in Seconds for Various Modes.

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

LRT mutant in MathComp: In this mutant, the last empty list `[::]` is removed from an auxiliary function used by an implementation of the merge sort algorithm:

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

In essence, mutation preserves the functional correctness of sorting. However, the complexity of the sort function changes from $O(n \log n)$ to $O(n^2)$. According to the author of the function (in personal communication), Georges Gonthier, "the 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." He concluded that "without the `[::]` placeholder the MathComp sort becomes two element-wise insertion sort."

BFT in Flocq: In this mutant, `false` is changed to `true` in the following function:

```
Definition shr_1 mrs :=
let '(Build_shr_record m r s) := mrs in
let s := orb r s in
match m with
| Zneg (xO p) ⇒
  Build_shr_record (Zneg p) false s
```

Although there are several lemmas about `shr_1` below the definition, none of them touch this particular match case. In fact, there are no lemmas at all about `Zneg` (negative integer) cases of `shr_1`. This indicates that `Zneg` cases in `shr_1` are unused elsewhere, and we found that they are actually assumed away implicitly by guards in lemmas.

Discussion. In addition to the live mutants, we also analyzed two killed mutants from every project by sampling uniformly at random; all were killed by a nearby proof (same file). PrettyParsing and TLC have the lowest mutation scores of all projects; 83.33% and 76.88%, respectively. We expected the utility libraries (Coq-std++, TLC, and StructTact) to have relatively low scores, due to the greater number of functions and datatypes than in more focused projects.

The relatively high score of Coq-std++, despite its size in terms of LOC, may indicate that most definitions are extensively specified. To corroborate this, the main author of Coq-std++ emphasized in personal communication that he consistently proves several lemmas about each new definition added to the library. The main author of TLC explained in personal communication that, in contrast to Coq-std++, key lemmas about TLC definitions are sometimes placed in other projects for reasons of convenience. The relatively low score of PrettyParsing likely stems from that its main theorem, that deserializing serialized "prettified" data gives correct results,

TABLE VII: Number of Definitions Found by Dependency Tools With Various Parameters.

Project	grep		defusage			Total
	$=1$	>1	$=0$	≤ 5	≤ 10	
ATBR	115	1662	443	2266	2544	2760
FCSL PCM	42	527	82	464	532	585
Flocq	26	221	32	187	229	257
Huffman	4	82	7	63	80	90
MathComp	1054	4946	804	4616	5397	6051
PrettyParsing	3	138	16	120	139	151
Bin. Rat. Numbers	21	234	43	290	329	379
Quicksort Compl.	11	255	30	243	275	296
Stalmarck	6	264	37	229	271	304
Coq-std++	193	645	134	664	789	869
StructTact	2	45	15	32	44	47
TLC	41	863	94	765	850	956

does not take into account most details on *how* prettification is done (through functions modified by live mutants).

4) RQ4: Comparison to Dependency Analysis: As mentioned in Section II-C, dependency analysis is used by proof engineers to analyze their verification projects, and may highlight some unused definitions similar to those we labeled DanglingDef. To enable comparing mutation proving with dependency analysis, we used the Coq `dpdgraph` plugin [23] to obtain, for each project, (1) a dependency graph of all definitions and lemmas, and (2) a list of the names of all definitions. We also extended `dpdgraph` to produce a tool dubbed `defusage` that counts edges *to* definitions in graphs.

As a simple baseline, we used `grep` to record the number of matches for each definition in each project's `.v` files. The first three columns in Table VII show the project name and number of definitions that had *exactly one* and *more than one* match, respectively, with `grep`. These can be compared to those in the last column, which show the total number of definitions. As a more robust alternative, we used `defusage` on the dependency graph of each project, with three thresholds in terms of number of incoming edges: 0 (unused), 5, and 10. Columns four to six in Table VII show the number of definitions *at or below* each threshold for all projects.

The large discrepancies between the second and fourth columns of Table VII indicate that the `grep` baseline is both unsound and incomplete; for some projects such as ATBR, it finds only a fraction of all unused definitions, while for, e.g., MathComp, it finds too many unused definitions. More importantly, *none* of the definitions changed by the live mutants we manually analyzed were included among the definitions in the second column (`grep` $= 1$). We conclude that the `grep` baseline is unusable for finding incompletely specified functions and datatypes.

While `defusage` produces sound and complete lists of unused definitions, the lists are typically long, and contain a large percentage of all definitions even with threshold 0 (e.g., for MathComp), making it hard to apply in practice. Among definitions changed by the live mutants we labeled with UnderspecifiedDef and DanglingDef, only 12 out of 63, all labeled DanglingDef, are found among those in column four ($= 0$). We conclude that mutation proving finds many more fundamental flaws in Coq verification projects than dependency analysis, and does so in a more informative, systematic, and less noisy way.

VI. Threats to Validity

External. Our results may not generalize to all Coq projects. To mitigate this threat, we chose popular projects that differ in size, number of proofs, and proof checking time. As our infrastructure builds on Coq 8.9, we could only use projects that work with this Coq version. We report results for a single hardware platform, and results may differ if experiments are run elsewhere. We ran all our experiments on two platforms, but we reported results only for one of them (more modern) due to space limitations. Although absolute numbers differ across platforms, our conclusions remain unchanged. We only analyzed a subset of killed and live mutants in our qualitative study. Our findings could differ if we had inspected a different set or more mutants. We mitigate this threat by systematically sampling mutants for inspection.

Internal. Our implementation of the tool and/or scripts may have bugs. To mitigate this threat, we performed extensive unit testing of our code. We also checked that results were the same across modes and that execution time differences were negligible across several runs. Finally, during our qualitative analysis, we validated the outcome of each mutant we studied.

Construct. Our work targets only Coq. Nevertheless, many mutation operators described in Section III-B, e.g., all operators in the Lists category, are applicable to projects using other proof assistants such as Lean [52] and Isabelle/HOL [53]. However, more research is needed to develop an extensive set of mutation operators and evaluate mutation proving for other proof assistants and deductive verification tools.

VII. Limitations and Future Work

Mutation operator design. We implemented and experimented with a mutation operator for changing the order of cases in a pattern matching expression, inspired by Le et al. [11]. However, mutants generated by this operator were nearly always killed immediately (stillborn), since Coq pattern matching branches tend to be completely unambiguous, and the strong type system does not permit leaving out matching cases. This illustrates the problem of defining general operators for Gallina, as opposed to operators using the standard library, e.g., addition for Peano arithmetic. A highly idiomatic Coq project may benefit from using specialized operators for the libraries it depends on.

Scope of mutation. We do not consider mutation of lemma statements or of Ltac proof scripts. The main reason is that we then would largely lose the analogy between mutation proving and mutation testing, since mutation of *test code* is not performed in the latter. Inductive predicates, which are a special form of inductive datatypes, are arguably borderline cases, but we included them for mutation based on their established interpretation as cut-free higher-order Prolog programs [13].

Equivalence and mutation scores. Mutation testing traditionally uses *functional equivalence* of programs in its definition of mutation score, which makes score calculation undecidable in general and usually necessitates using heuristics to filter out equivalent mutants [54]. In contrast, this equivalence is only one of many that may be considered when defining mutation score for Coq projects. We implemented checking of syntactical equality at the vernacular level, which preserves proofs but is highly discriminating. It is also possible to define and compute scores using Coq's least discriminating decidable notion of equivalence that always preserves proofs, *convertibility* [55], or using a project-specific equivalence. However, checking convertibility is costly, and only two of the live mutants we labeled SemanticallyEq were convertible.

Alternative mutation approaches. While our operators are defined and applied at the level of vernacular syntax, SERAPI also supports serialization of data added during the *elaboration phase* [56] of type checking in Coq. Additional operators can potentially use this information to perform sophisticated type-preserving changes to Coq definitions. However, such operators may intuitively no longer capture mistakes that are made by proof engineers, which our operators aim at doing.

VIII. Related Work

Since, to our knowledge, ours is the first evaluation of mutation analysis for proof assistants, we contrast with mutation analysis in similar settings and other analysis techniques.

Mutation testing of functional programs. Le et al. [11], [57] implemented a mutation testing framework for Haskell called MuCheck, which applies mutation operators nondeterministically at the level of abstract syntax trees. Cheng et al. [12] used MuCheck to evaluate different types of test coverage for Haskell programs. Duregård [58] proposed a *black-box* approach to mutation testing of Haskell code, on top of the QuickCheck framework. The function under test must be an instance of a specific type class that allows it to be mutated (without modifying it in-place). Braquehais and Runciman [59] presented a Haskell framework, FitSpec, that uses mutation testing to measure adequacy of sets of properties specified in property-testing frameworks such as QuickCheck. FitSpec takes a black-box view of mutations, and uses instance enumeration to produce mutants.

On one hand, black-box mutation can be applied in a wider context than operator-based mutation, e.g., to functions associated with native code. On the other hand, black-box mutation sometimes requires defining explicit functions that return mutants. While black-box mutation can be implemented in Coq, we believe the purity of Gallina makes its advantages modest compared to operator-based mutation.

Mutation of specifications. We took inspiration from Groce et al. [8], who use mutation analysis to improve the process of verification based on model checking. Ball and Kupferman [9] consider the concept of *vacuity* in verification and testing, which can be established through mutation of systems and their specifications. Mutation proving is intuitively similar to, but more general than, their notion of vacuity in software checking. Efremidis et al. [60] presented a mutation framework for Prolog with operators reminiscent of ours.

Analysis and testing in proof assistants. Berghofer and Nipkow [61] first considered random testing to assist users of Isabelle/HOL to specify and verify programs. Bulwahn [22] subsequently improved the Isabelle testing facilities. A testing framework for Coq, dubbed QuickChick, was proposed by Paraskevopoulou et al. [26], and Lampropoulos and Pierce [10] describe mutation testing in that framework. Blanchette and Nipkow [24] presented a counterexample generator for Isabelle/HOL. Cruanes and Blanchette [25] later presented a general tool, Nunchaku, for counterexample generation, and showed how to adapt it to dependent type theories like Coq's. Johansson [28] proposed a tool for theory exploration in Isabelle/HOL called Hipster, which attempts to prove interesting facts from a given set of definitions.

Testing and generation as in QuickChick and Nunchaku can analyze specific functions and datatypes to find problematic inputs, but lack the connection to proofs that MCOQ has. Moreover, the above techniques do not consider alternative "worlds" with different definitions, and are thus largely complementary to mutation proving. For example, Hipster could be applied to mutants to reveal facts that are consequences of alternative definitions generated by MCOQ.

IX. Conclusion

We proposed mutation proving, a technique for analyzing verification projects that use proof assistants. We implemented our technique for the Coq proof assistant in a tool dubbed MCOQ. MCOQ applies a set of mutation operators to Coq definitions of functions and datatypes, inspired by our experience and operators previously defined for functional programming languages. MCOQ then checks proofs of lemmas affected by operator application. To make our technique feasible in practice, we implemented several optimizations in MCOQ such as parallel proof checking. We applied MCOQ to 12 medium and large scale Coq projects, and recorded whether proofs passed or failed when applying different mutation operators. We then qualitatively analyzed the failed proofs, finding several examples of incomplete specifications. Moreover, our work has already had significant impact on Coq tooling, and our tool helped to uncover a bug in Coq itself. We believe that MCOQ can be extended in a number of ways and already be useful for many practical tasks. We are looking forward to see MCOQ used by proof engineers for improving the quality of their verification projects and by researchers for evaluating new proof engineering techniques.

Acknowledgments

We thank the anonymous reviewers for their comments and Ina Schaefer for shepherding this paper; Arthur Charguéraud, Georges Gonthier, Farah Hariri, Kush Jain, Robbert Krebbers, Pengyu Nie, Zachary Tatlock, James R. Wilcox, and Théo Zimmermann for their feedback. This work was partially supported by the US National Science Foundation under Grant Nos. CCF-1652517 and CCF-1704790.

References

[1] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: An analysis and survey," *Advances in Computers*, vol. 112, pp. 275–378, 2019.

[2] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Symposium on Operating Systems Principles*, 2009, pp. 207–220.

[4] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson, "Planning for change in a formal verification of the Raft consensus protocol," in *Certified Programs and Proofs*, 2016, pp. 154–165.

[5] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock, "QED at large: A survey of engineering of formally verified software," *Foundations and Trends in Programming Languages*, vol. 5, no. 2-3, pp. 102–281, 2019.

[6] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," in *European Conference on Computer Systems*, 2017, pp. 328–343.

[7] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.

[8] A. Groce, I. Ahmed, C. Jensen, P. E. McKenney, and J. Holmes, "How verified (or tested) is my code? Falsification-driven verification and testing," *Automated Software Engineering*, vol. 25, no. 4, pp. 917–960, 2018.

[9] T. Ball and O. Kupferman, "Vacuity in testing," in *Tests and Proofs*, 2008, pp. 4–17.

[10] L. Lampropoulos and B. C. Pierce, "QuickChick Interface," 2018. [Online]. Available: https://softwarefoundations.cis.upenn.edu/qc-current/QuickChickInterface.html

[11] D. Le, M. A. Alipour, R. Gopinath, and A. Groce, "MuCheck: An extensible tool for mutation testing of Haskell programs," in *International Symposium on Software Testing and Analysis*, 2014, pp. 429–432.

[12] Y. Cheng, M. Wang, Y. Xiong, D. Hao, and L. Zhang, "Empirical evaluation of test coverage for functional programs," in *International Conference on Software Testing, Verification, and Validation*, 2016, pp. 255–265.

[13] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[14] Coq Team, "Coq manual: Syntax extensions and interpretation scopes," 2019. [Online]. Available: https://coq.inria.fr/distrib/V8.9.0/refman/user-extensions/syntax-extensions.html

[15] ——, "Coq manual: Utilities," 2019. [Online]. Available: https://coq.inria.fr/distrib/V8.9.0/refman/practical-tools/utilities.html

[16] A. Celik, K. Palmskog, and M. Gligoric, "iCoq: Regression proof selection for large-scale verification projects," in *International Conference on Automated Software Engineering*, 2017, pp. 171–182.

[17] A. Groce, J. Holmes, D. Marinov, A. Shi, and L. Zhang, "An extensible, regular-expression-based tool for multi-language mutant generation," in *International Conference on Software Engineering, Demo*, 2018, pp. 25–28.

[18] E. J. Gallego Arias, "SerAPI: Machine-Friendly, Data-Centric Serialization for Coq," MINES ParisTech, Tech. Rep., 2016. [Online]. Available: https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408

[19] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part I," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[20] E. J. Gallego Arias, "Coq issue #9204," 2018. [Online]. Available: https://github.com/coq/coq/issues/9204

[21] E. Tassi, "Coq pull request #9206," 2018. [Online]. Available: https://github.com/coq/coq/pull/9206

[22] L. Bulwahn, "The new Quickcheck for Isabelle: Random, exhaustive and symbolic testing under one roof," in *Certified Programs and Proofs*, 2012, pp. 92–108.

[23] A. Pacelet and Y. Bertot, "coq-dpdgraph," 2019. [Online]. Available: https://github.com/Karmaki/coq-dpdgraph

[24] J. C. Blanchette and T. Nipkow, "Nitpick: A counterexample generator for higher-order logic based on a relational model finder," in *International Conference on Interactive Theorem Proving*, 2010, pp. 131–146.

[25] S. Cruanes and J. C. Blanchette, "Extending Nunchaku to dependent type theory," in *International Workshop on Hammers for Type Theories*, vol. 210, 2016, pp. 3–12.

[26] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce, "Foundational property-based testing," in *International Conference on Interactive Theorem Proving*, 2015, pp. 325–343.

[27] Z. Chen, L. O'Connor, G. Keller, G. Klein, and G. Heiser, "The Cogent case for property-based testing," in *Workshop on Programming Languages and Operating Systems*, 2017, pp. 1–7.

[28] M. Johansson, "Automated theory exploration for interactive theorem proving," in *International Conference on Interactive Theorem Proving*, 2017, pp. 1–11.

[29] D. Delahaye, "A tactic language for the system Coq," in *Logic for Programming and Automated Reasoning*, 2000, pp. 85–95.

[30] OCaml Labs, "PPX," 2017. [Online]. Available: http://ocamllabs.io/doc/ppx.html

[31] E. J. Gallego Arias, B. Pin, and P. Jouvelot, "jsCoq: Towards hybrid theorem proving interfaces," in *Workshop on User Interfaces for Theorem Provers*, 2017, pp. 15–27.

[32] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[33] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," in *Symposium on Principles of Programming Languages*, 1980, pp. 220–233.

[34] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[35] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *International Symposium on Foundations of Software Engineering*, 2014, pp. 654–665.

[36] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults," in *International Conference on Software Engineering*, 2018, pp. 537–548.

[37] A. Chlipala, "Ltac anti-patterns," 2019. [Online]. Available: http://adam.chlipala.net/cpdt/html/Large.html

[38] H. Coles, "PIT mutation testing," 2010. [Online]. Available: http://pitest.org

[39] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *International Symposium on Software Testing and Analysis*, 2014, pp. 433–436.

[40] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *International Symposium on Software Testing and Analysis*, 2014, pp. 315–326.

[41] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Mutation reduction strategies considered harmful," *Transactions on Reliability*, vol. 66, no. 3, pp. 854–874, 2017.

[42] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *Transactions on Software Engineering*, 2018.

[43] R. Gopinath, C. Jensen, and A. Groce, "Topsy-Turvy: A smarter and faster parallelization of mutation analysis," in *International Conference on Software Engineering, Demo*, 2016, pp. 740–743.

[44] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," in *International Symposium on Software Testing and Analysis*, 2017, pp. 295–306.

[45] L. Chen and L. Zhang, "Speeding up mutation testing via regression test selection: An extensive study," in *International Conference on Software Testing, Verification, and Validation*, 2018, pp. 58–69.

[46] K. Palmskog, A. Celik, and M. Gligoric, "piCoq: Parallel regression proving for large-scale verification projects," in *International Symposium on Software Testing and Analysis*, 2018, pp. 344–355.

[47] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 249–258.

[48] E. J. Gallego Arias, "SerAPI: The Coq Se(xp)rialized Protocol," 2019. [Online]. Available: https://github.com/ejgallego/coq-serapi

[49] G. Gonthier and A. Mahboubi, "An introduction to small scale reflection in Coq," *Journal of Formalized Reasoning*, vol. 3, no. 2, pp. 95–152, 2010.

[50] J. Mendez, "jsexp," 2019. [Online]. Available: https://github.com/julianmendez/jsexp

[51] B. Barras, C. Tankink, and E. Tassi, "Asynchronous processing of Coq documents: From the kernel up to the user interface," in *International Conference on Interactive Theorem Proving*, 2015, pp. 51–66.

[52] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The Lean theorem prover (system description)," in *International Conference on Automated Deduction*, 2015, pp. 378–388.

[53] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[54] M. Papadakis, M. Delamaro, and Y. Le Traon, "Mitigating the effects of equivalent mutants with mutant classification strategies," *Science of Computer Programming*, vol. 95, pp. 298–319, 2014.

[55] Coq Team, "Coq manual: Conversion rules," 2019. [Online]. Available: https://coq.inria.fr/distrib/V8.9.0/refman/language/cic.html#conversion-rules

[56] L. de Moura, J. Avigad, S. Kong, and C. Roux, "Elaboration in dependent type theory," *CoRR*, vol. abs/1505.04324, 2015.

[57] D. Le, M. A. Alipour, R. Gopinath, and A. Groce, "Mutation testing of functional programming languages," Oregon State University, Tech. Rep., 2014.

[58] J. Duregård, "Automating black-box property based testing," Ph.D. dissertation, Chalmers University of Technology, 2016.

[59] R. Braquehais and C. Runciman, "FitSpec: Refining property sets for functional testing," in *Haskell Symposium*, 2016, pp. 1–12.

[60] A. Efremidis, J. Schmidt, S. Krings, and P. Körner, "Measuring coverage of Prolog programs using mutation testing," in *Functional and Constraint Logic Programming*, 2019, pp. 39–55.

[61] S. Berghofer and T. Nipkow, "Random testing in Isabelle/HOL," in *International Conference on Software Engineering and Formal Methods*, 2004, pp. 230–239.