mCoq: Mutation Analysis for Coq Verification Projects

Kush Jain  
kjain14@utexas.edu  
The University of Texas at Austin

Karl Palmskog  
palmskog@acm.org  
KTH Royal Institute of Technology

Ahmet Celik  
celik@fb.com  
Facebook, Inc.

Emilio Jesús Gallego Arias  
e@x80.org  
Équipe πr², Inria, IRIF, Univ. de Paris

Milos Gligoric  
gligoric@utexas.edu  
The University of Texas at Austin

1 INTRODUCTION

Software developed and formally verified using proof assistants, such as Coq, can provide trustworthiness beyond that of software developed using traditional programming languages and testing practices. However, guarantees from formal verification are only as good as the underlying definitions and specification properties. If properties are incomplete, flaws in definitions may not be captured during verification, which can lead to unexpected system behavior and failures. Mutation analysis is a general technique for evaluating specifications for adequacy and completeness, based on making small-scale changes to systems and observing the results. We demonstrate mCoq, the first mutation analysis tool for Coq projects. mCoq changes Coq definitions, with each change producing a modified project version, called a mutant, whose proofs are exhaustively checked. If checking succeeds, i.e., the mutant is live, this may indicate specification incompleteness. Since proof checking can take a long time, we optimized mCoq to perform incremental and parallel processing of mutants. By applying mCoq to popular Coq libraries, we found several instances of incomplete and missing specifications manifested as live mutants. We believe mCoq can be useful to proof engineers and researchers for analyzing software verification projects. The demo video for mCoq can be viewed at: https://youtu.be/QuhgpF7dNo.

2 TECHNIQUE AND IMPLEMENTATION

In this section, we briefly describe our mutation analysis technique, which we call mutation proving [3], that mCoq implements. We then explain the main mCoq components and workflow.

Mutation proving is based on exhaustively applying mutation operators to Coq projects. When an operator is successfully applied...
to a source file, it generates a mutant where a Coq definition is different from before. For example, an operator can change addition of natural numbers to subtraction, or remove the head of a list. Note that in analogy with mutation testing, which does not mutate tests, we do not change specification properties or proofs. If all proofs associated with a mutant are successfully checked, the mutant is reported live; otherwise, it is reported killed. A project’s mutation score is defined as the percentage of killed mutants out of all synthetically distinct generated mutants. A low mutation score intuitively, but not unambiguously, indicates that specifications may be incomplete or missing for some definitions.

mCoq implements mutation proving by first serializing all Coq code to S-expressions [15] (sexps) using the SerAPI library, then transforming the sexps pinpointed by mutation operators, and finally deserializing the results and checking all associated proofs. Fig. 1 shows the key components of mCoq, which are essentially unchanged from the initial version. The sercomp command-line tool, which we developed and is included in SerAPI, handles (de)serialization. The standard coqc tool proof-checks source files. Finally, the QMutator tool that we wrote in Java transforms the sexps.

Checking a Coq proof can be time-consuming, e.g., due to invocations of arithmetic constraint solvers in proof scripts. Our optimized mode for mCoq performs incremental analysis that avoids unnecessary (de)serialization and parallelizes checking of each individual mutant. In the evaluation of our previous tool version, we found that this mode consistently outperforms other parallel modes. We also include a basic mode, called Default, which naively (de)serializes and checks all files for every mutant.

To collect mutant data for reporting, we process the log files generated by QMutator using Python, computing information on killed and live mutants, mutation scores, and execution time. We then generate HTML reports in a format inspired by the format of the JaCoCo library [11], as explained in more detail in Section 3.

3 TOOL INSTALLATION AND USAGE

This section describes how users can install and integrate mCoq into their workflows.

3.1 Installation

The first installation step is to install Coq 8.10 and SerAPI 0.7. We recommend installing these dependencies via the OCaml-based package manager OPAM [17], version 2.0.5 or later:

```
$ opam update
$ opam install coq.8.10.2 coq-serapi.8.10.0+0.7.0
```

Then, users should clone the mCoq repository on GitHub:

```
$ git clone https://github.com/EngineeringSoftware/mcoq.git
$ cd mcoq && git checkout v1.0
```

Additionally, we assume JDK 8 or later and the Gradle build system [8] are installed. Finally, our entry script (mcoq.py) for the tool requires Python 3 to be available on the system.

3.2 Usage

After installation, users can interact with mCoq via mcoq.py, e.g.,

```
$ ./mcoq.py --help
```

Applying mCoq to a Coq project requires that the project code (1) lives in a Git-based repository and (2) contains a _CoqProject file in the root directory listing all relevant Coq source files, in the format used by the coq_makefile tool [5]. If these requirements are met, no changes are required to use mCoq on a project.

For example, running the following command in the mcoq directory fetches the Coq project StructTact at revision 82a85b7 from its GitHub repository and mutates it using two parallel threads:

```
$ ./mcoq.py --project StructTact --sha 82a85b7 --threads 2
```

This command runs mCoq using the default configuration, meaning, e.g., that all mutation operators are enabled. Table 1 lists and describes all arguments currently available for the tool. Note that changing the mode via the option --mutator is provided to enable further research on efficient mutation analysis for Coq; we do not expect this option to be used for other purposes.

When the execution of the above command has completed, the result is a log file and an HTML report. The log file contains detailed information about the entire execution (trace), generated mutants, and execution time and outcome (killed or live) for each mutant. The log is used both for debugging the tool (when necessary) and for generating the more user-friendly report. The report is automatically generated in the reports directory and can be displayed with any modern web browser. We provide an example report at: https://cozy.ece.utexas.edu/mcoq/report.
3.3 Reports in Detail

Figure 2 shows two screenshots of the HTML report generated for the StructTact project, which is one of the projects we used in our evaluation (see Section 4). To design the report pages generated by mCoq, we initially took inspiration in JaCoCo [11], a popular code coverage tool for Java projects. However, JaCoCo’s task is simpler: it only provides a binary flag for each line (a line is colored green if covered and red if not covered). For mutation analysis, we have to report (potentially) multiple mutants per line, the operator that generated each mutant, and the outcome of the analysis.

Figure 2(a) gives an overview of the outcome of the mutation analysis. Specifically, the top part of the figure shows the SHA of the project for which the report was generated, time when the report was generated, Coq version, mCoq mode, total running time (in milliseconds), total time to run sercomp, number of mutants that were killed due to timeout, number of syntactically equivalent mutants, and total number of source files in the project. The bottom part of the figure shows the table that includes the key metrics for mutation analysis: mutation score, number of generated mutants, and number of killed mutants. We show these metrics for the entire project in the first row ("All files"). Subsequent rows show one file per line (sorted alphabetically). A user has the option to expand "Mutation Operator" and see the score per operator; this is shown in the figure for the file Before.v. We use green, red, and yellow, for 100%, 0%, and any other mutation score, respectively; gray with no number is used for operators that generated no mutant.

If a user clicks on the file Before.v, she sees the page in Figure 2(b) displaying the file contents, including line numbers and syntax highlighting. Additionally, lines that contain at least one generated mutant are highlighted; green color (e.g., line 9 in the example) is used for those lines where all mutants are killed and red (e.g., line 17 in the example) is used for those lines where at least one mutant is live. If a user would like to see the details about mutants for a specific line, she can click on "View Operators", which will show one line per operator for operators that generated at least one mutant (see lines between 17 and 18 in the example).

4 EVALUATION

To evaluate mCoq, we performed mutation analysis of several popular Coq libraries. We report the number of generated, killed, and live mutants. We also summarize the speedup for our optimized mutation analysis mode compared to the basic mode. Finally, we summarize our findings from manual inspection of live mutants.

The numbers that we report are obtained with the latest version of mCoq compatible with Coq 8.10. However, we also discuss the numbers we obtained with the previous version of the tool compatible with Coq 8.9 [3]; we performed inspection of live mutants found with the previous version of mCoq.

4.1 Coq Projects Under Study

Table 2 lists the name of every project we used in the evaluation, along with the revision SHA we used in the experiments. If SHAs used for the previous mCoq evaluation were not compatible with Coq 8.10, we used more recent SHAs. The table also lists the number of generated, killed, and live mutants, as well as mutation score.

4.2 Summary of Results

We ran all experiments for our evaluation on a 6-core Intel Core i7-8700 CPU @ 3.20GHz machine with 64GB of RAM, running Ubuntu Linux 18.04.1 LTS. We limit the number of parallel threads to be at or below the number of physical CPU cores.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Mutation Operator</th>
<th>Mutation Score</th>
<th>Generated</th>
<th>Killed</th>
</tr>
</thead>
<tbody>
<tr>
<td>All files</td>
<td>Total for structtact</td>
<td>154</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Before.v</td>
<td>All (expand)</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>LeftEmptyConcat</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveSuccessorFromApplication</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recorder</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RecorderIfExpressions</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveFalseWithTrue</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveIntWithEmptyList</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveIntWithHead</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveIntWithTail</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemovePlusWithMinus</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveSuccessorWithZero</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveTrueWithFalse</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveZeroWithOne</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveSuccessorsCosts</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RightEmptyConcat</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BeforeAll.v</td>
<td>All (expand)</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BookAll.v</td>
<td>All (expand)</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: HTML report automatically generated by mCoq for StructTact project used in our evaluation.
We demonstrated TLC and 97.39% (FCSL PCM), and is 92.37% on average across (out of 361 live mutants we found in our earlier evaluation [3]) and all projects. Note that these numbers only differ slightly from our original evaluation [3] due to the update to Coq 8.10 (from Coq 8.9) and updates in the projects (for those projects that we have to perform and incremental checking substantially outperformed all other modes, in our latest experiment. Our results confirm that the most original evaluation. Finally, we manually inspected 74 live mutants determined that 33 indicate incomplete operators based on the way a project uses libraries is an exciting avenue. Our design of the initial set of mutation operators was inspired by our extensive experience with Coq and prior work on mutation analysis for functional languages [13]. Designing specialized operators based on the way a project uses libraries is an exciting future direction. mCoQ targets only projects written in Coq; more research is needed to design and evaluate an extensive set of mutation operators for other proof assistants, such as Lean [6] and Isabelle/HOL [16]. We also plan to explore other ways to detect semantically equivalent mutants, e.g., convertibility in Coq [4] and further analyze killed mutants [10, 20]. Finally, mCoQ currently applies operators on Coq abstract syntax obtained immediately after parsing. An alternative approach is to mutate representations available at the later elaboration phase of type checking in Coq. We plan to explore this direction in the future.

### 4.3 Limitations and Future Work

Our design of the initial set of mutation operators was inspired by our extensive experience with Coq and prior work on mutation analysis for functional languages [13]. Designing specialized operators based on the way a project uses libraries is an exciting future direction. mCoQ targets only projects written in Coq; more research is needed to design and evaluate an extensive set of mutation operators for other proof assistants, such as Lean [6] and Isabelle/HOL [16]. We also plan to explore other ways to detect semantically equivalent mutants, e.g., convertibility in Coq [4] and further analyze killed mutants [10, 20]. Finally, mCoQ currently applies operators on Coq abstract syntax obtained immediately after parsing. An alternative approach is to mutate representations available at the later elaboration phase of type checking in Coq. We plan to explore this direction in the future.

### 5 CONCLUSION

We demonstrated mCoQ, the first tool for mutation analysis of Coq projects, useful for detecting incomplete specifications. mCoQ takes a Coq project as an input, generates mutants, finds what mutants are killed (or live), and generates a report. Our original modifications to Coq and SerAPI needed to perform mutation analysis, have been accepted by its developers into the latest versions of these projects. This enables a smooth installation and integration of mCoQ into existing workflows. Considerable care has been taken to ensure smooth evolution of the functionality in Coq and SerAPI which mCoQ relies on; we expect only minor effort to maintain mCoQ over time as Coq itself evolves. For our evaluation, we used mCoQ to perform mutation analysis on several popular Coq libraries and found many incomplete specifications. We believe that mCoQ is ready for a wider use in software verification projects by both proof engineers and researchers.

### ACKNOWLEDGMENTS

We thank Marinela Parovic for earlier contributions to the project. We also thank Arthur Charguéraud, Georges Gonthier, Farah Harri, Catalin Hritcu, Robbert Krebbers, Pengyu Nie, Zachary Tatlock, James R. Wilcox and Théo Zimmermann for their feedback on this work. This work was partially supported by the US National Science Foundation under Grant No. CCF-1652517.

### REFERENCES


