# Incremental Specification Mining
# CS 6156

Kate Meuse, Jan-Paul Ramos

## 1 ABSTRACT

Specification mining is fundamental to verifying area-specific code invariants with runtime verification. However, the computational overhead increases monstrously for non-trivially sized repositories (> `100 MB`). Similar problems are found in runtime monitoring, for which evolutionary-aware tools have been developed in an attempt to ameliorate complexity. However, in the context of specification mining, current evolution-aware analysis just compounds complexity further, due to the need for re-mining specifications at each *evolution* in a repository's lifespan. We propose the first model for an *incremental specification mining* pipeline (ISM). By making use of ISM, specification miners can reap the benefits from an evolutionary aware system while getting rid of unnecessary overhead. We also present alternative ideas for integrating ISM into existing specification mining frameworks.

## 2 INTRODUCTION

Techniques for runtime monitoring, such as *Monitoring-Oriented Programming (MOP)*, monitors software against formally verified correctness properties. Such techniques have been put into practice with deployable tools for large-scale monitoring, i.e., JavaMOP [3]. JavaMOP adds instrumentation in software wherever monitored events take place, and the instrumentation executes the code generated from a set of formal properties. While useful, one of the main concerns with MOP-dependent techniques is the lack of automation for generating formal specifications. These specifications take non-trivial intellectual power and time to come up with, and with manual analysis, defeat the purpose of using lightweight over formal methods which would require a similar amount of intellectual power to develop.

A popular technique–and the focus of this report–for aiding developers in producing a set of salient specifications is *specification mining* (SM). Specification miners follow the maxim "Common behavior is correct behavior. [2]" Although there are still problems with the latest state-of-the-art specification miners (detailed below), we believe that the probabilistic nature of SM is a strong contender for alleviating lightweight overhead brought by manually coming up with formal specifications for a runtime monitor.

One of the key advantages of SM is that it provides benefits for redundant programming [2]. Exposed to large codebases, a miner can collect and summarize API protocols, and probabilistically determine which protocol is correct for various versions of a codebase. However, due to this exhaustive approach, specification miners don't scale well in the context of evolutionary-aware monitoring systems. These systems make use of the accumulated benefits demonstrated by [6], [7], to monitor multiple versions of evolving software rather than a single version[1]. Specification miners are currently designed to instrument and mine specifications from a given repository, naively identifying each snapshot of an evolutionary repository as independent projects. This requires re-mining specifications for each snapshot of a repository, exponentially increasing the overhead when paired with tools to ameliorate the already existing problems in MOP-tools.

We propose a specification mining analog to evolutionary-aware monitoring: evolutionary-aware specification mining. Concretely, this is targeted towards *incremental specification mining* (ISM). ISM uses the same techniques from evolutionary-aware program monitoring to identify when a given snapshot for a program qualifies as an *evolution*, and will consequently only mine for all evolutions for the entire lifetime of a repository. ISM also addresses questions in the *usefulness* of certain specifications throughout the lifetime of a repository, highlighting *unique* specifications for each snapshot, as it will not re-mine specifications for the same traces.

Further, we present a background on evolutionary-aware systems and techniques, in addition to current tools that attempt evolutionary-aware specification mining 3. In section 4, we talk about our general approach, and in section 5 how these approaches were implemented as several shell scripts that made use of existing specification miners and evolutionary-aware tools. In section 6 we present preliminary results from comparing a naive run on an open-source repository with the evolutionary-aware approach. Finally, in section 7 we discuss future directions to ISM.

## 3 BACKGROUND

The main extension for MOP using an evolutionary-aware approach is eMOP [4]. eMOP uses regression test selection (RTS), regression property selection (RPS), and regression monitor selection (RMS) for studying and monitoring the evolution of a repository throughout multiple versions of software. For gathering knowledge of the changes between two versions of software, eMOP uses RTS, which has the added benefit that running fewer tests decreases runtime overhead and triggers fewer violations! The key tool we use for RTS analysis is therefore STARTS [5]. STARTS is efficient because it does not require code instrumentation to find impacted tests, therefore we don't need a trade-off for the SM instrumentation we'll encounter later on. In addition, STARTS is a Maven plugin, allowing for easy integration into our set of open-source projects.

From what we have gathered thus far, this is the first approach toward an applicable ISM. Previous work in evolutionary-aware SM is particularly desolated, with the exception of the SPECMATE project [1]. SPECMATE generates incremental specifications for a set of relevant properties. However, this project targets mutation analysis rather than the natural evolution of industrial-sized programs. This is useful for test case generation, in generating additional runs to explore the execution space. However, we believe an evolutionary-aware ISM technique can be implemented more comfortably into existing evolutionary-aware frameworks, such as

[1]We define an *evolution* as a difference between in stages in the traces for a given repository's test classes.

eMOP, whereas SPECMATE is coupled too closely with generating test suites.

## 4 APPROACH

At a high level, we are just assisting specification mining with an oracle, given by STARTS, which determines which files have changed.

### 4.1 Persistent Specifications

To find the persistent specifications, we use a selection of specification miners to obtain program traces of various open-source repositories that use Maven as a build system. We then filter the specs by the number of commits they appear in.

### 4.2 Regression Test Selection

STARTS allows us to identify which test classes have been affected by a given evolution of the repository. This is particularly helpful because we can easily pipe the test classes into `methodtracer.jar` (a program for generating traces) in the following manner:

```
mvn test ${SKIPS} -DargLine="-javaagent:<Script>/
methodtracer.jar=all-tests@trace.include=<Evolution Traces>;instrument.exclude=<Non Evolution>"
```

We exclude the test classes which are not selected by STARTS by comparing a global array for all test classes identified at the first iteration, and comparing which ones are left out of the selected analysis at each snapshot. Snapshots that are empty (hence no evolution) are left out of the trace generation, therefore we are left with a significantly lesser amount of traces and SM significantly decreases in overhead.

## 5 TECHNIQUES

We began the semester by learning to run two main specification miners, Javert[2] and BDD. This required composing a Dockerfile that could run interactively as a shell for these miners, as they do not use current package versions and require careful setup. The Dockerfile is built on Ubuntu and includes Python 3.8 to run our scripts and Java 8, which was able to support the miners. Some repositories depended on different versions of Java, so we include additional JDKs and modify the default Java on-the-fly.

Both Javert and BDD run on Java bytecode traces. We **significantly** modified a script provided by Ayaka to work on multiple code versions without cloning the repository unnecessarily. The code versions were described in project files, which contained lists of Git URLs, names, and commit SHAs. We generated project files using a Python script that pulls a specified number of commit hashes. Using those, we could generate basic traces.

To analyze persistent specifications, we created another script to record which specifications were present in which commits. We did this by creating a unique hash for every specification and entering the commit numbers where that specification was found into a dictionary. To obtain persistent or nearly-persistent specs, we simply filtered by length. We then compare the number of persistent specs for intervals of commits to see how the number of persistent specs reduces over time. See Figure 1 for the overview.

To continue into our RTS analysis, we needed to integrate STARTS into this process(2). We limit the program traces we obtain to only

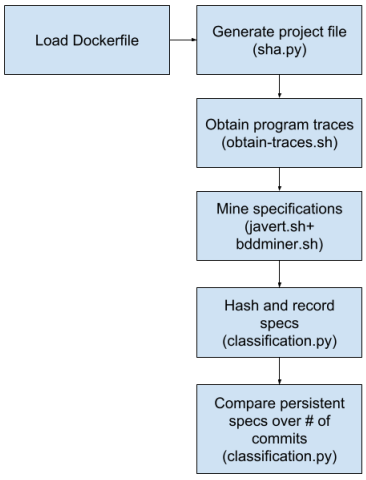


**Figure 1: Flowchart of persistent spec filtering**

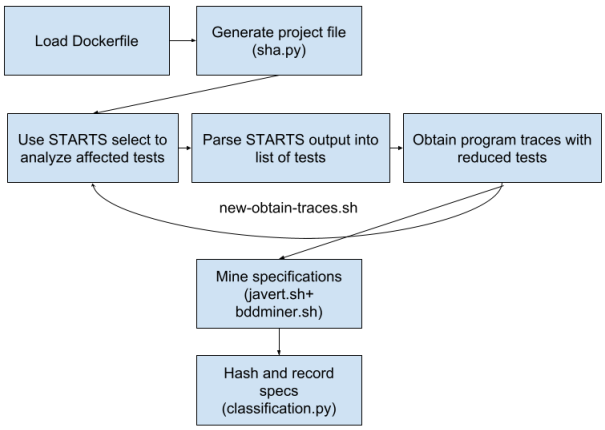


**Figure 2: Flowchart of STARTS filtered mining**

those created by the affected tests returned by STARTS(using the select option). We pass the parsed program arguments from out STARTS output to the method tracer JAR.

## 6 RESULTS

The revised program tracer and specification combination significantly reduced the number of produced specs. Frequently, there will be very minimal additional specs generated due to a small change in documentation or configuration file. While there used to be hundreds of thousands of consistent commits for even small

repositories, there are almost no persistent commits to no peristent commits depending on the repository and the breadth of their testing files. If the repository keeps all of their tests in one file, our analysis is obviously less effective.

### 6.1 Issues

There were several issues throughout the project in regard to our computers crashing, running our of memory, and running out of space. We could not get any help from Cornell IT with server space, therefore our benchmarks are made up of relatively small repositories.

When setting up STARTS, there were VM binding issues for detecting unconditionally starting the agent in Java 9. If not already, we recommend that the following caveat be mentioned in the STARTS repository:

```
export JDK_JAVA_OPTIONS=-Djdk.attach.allowAttachSelf=true
when raising the error that Java Agent Could Not Be Attached
```

## 7 FUTURE WORK

There are several optimizations that can be taken place to further ameliorate the runtime overhead of ISM. In particular, the integration of ISM with a system such as eMOP which takes into account evolutionary aware analysis using RPS in addition to RTS. While RTS is beneficial for smaller repositories, it lacks support for allowing our pipeline to generate traces when test classes fail, ignoring a significant portion of our traces. **The codebase for our current work can be found at https://github.com/cyankaet/spec-miners**

## REFERENCES

[1] URL: https://cordis.europa.eu/projects.
[2] Mark Gabel and Zhendong Su. "Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces". In: SIGSOFT '08/FSE-16. Atlanta, Georgia: Association for Computing Machinery, 2008, pp. 339–349. ISBN: 9781595939951. DOI: 10.1145/1453101.1453150. URL: https://doi.org/10.1145/1453101.1453150.
[3] Dongyun Jin et al. "JavaMOP: Efficient parametric runtime monitoring framework". In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 1427–1430. DOI: 10.1109/ICSE.2012.6227231.
[4] Owolabi Legunsen, Darko Marinov, and Grigore Rosu. "Evolution-Aware Monitoring-Oriented Programming". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. 2015, pp. 615–618. DOI: 10.1109/ICSE.2015.206.
[5] Owolabi Legunsen, August Shi, and Darko Marinov. "STARTS: STAtic regression test selection". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 949–954. DOI: 10.1109/ASE.2017.8115710.
[6] Francesco Logozzo et al. "Verification modulo Versions: Towards Usable Verification". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 294–304. ISBN: 9781450327848. DOI: 10.1145/2594291.2594326. URL: https://doi.org/10.1145/2594291.2594326.
[7] Lingming Zhang et al. "Regression Mutation Testing". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. Minneapolis, MN, USA: Association for Computing Machinery, 2012, pp. 331–341. ISBN: 9781450314541. DOI: 10.1145/2338965.2336793. URL: https://doi.org/10.1145/2338965.2336793.