December 2022

Validating Software Functionality Across Combinations of Runtime Configurations

Matt Kenison

Justin Bagwell

Tylor Sampson

Mike Meade

Validating Software Functionality Across Combinations of Runtime Configurations

ABSTRACT

As the number of configurable attributes of software under test grows and the cardinality of those attributes increases, the efficiency of software verification rapidly declines. This disclosure describes techniques to determine test coverage gaps in multidimensional configuration spaces of substantial cardinality by enabling a software developer to define a set of rules associated with a feature and by forming a Boolean algebra over a coverage matrix that defines the required test coverage. Test coverage is tracked and presented on a dashboard. Release can be blocked automatically if the test coverage gap is beyond a threshold. Based on the coverage matrix, untested combinations of configurations can be detected and individual compatibility tests to verify functionality can be designed. By dynamically maintaining the allowed values of the configuration space, intelligent prioritization of coverage becomes possible. Software testability is sustained in the face of exponential dimensional growth.

KEYWORDS

- Software testing
- Configurable attribute
- Dimensional cardinality
- Test coverage
- Coverage matrix
- Coverage tracking
- Release blocking
- Integration test
- Combinatorial testing

BACKGROUND

Software developers often have difficulty verifying that product features have been successfully tested across the combination of possible configurations under which the software is expected to run. As the number of configurable attributes (dimensions) grows and the cardinality of those combinations increases, the efficiency of software verification rapidly declines. There is often no efficient way to verify that each required configuration has an associated test pass in which those dimensions are present. It is not uncommon for commercial software to have thousands of dimensions, each dimension assuming one of hundreds of possible values. Testing the cartesian product of all values is computationally infeasible. For complex software developed and maintained by multiple teams, teams are required to maintain their tests against all other values even as the set of allowed values for each configuration changes.

While certain test frameworks support parameterization of one or more inputs and can generate a Cartesian matrix of combinations of configurations, the resulting parameters run tests individually with the given combination and have to be defined before running the test. No mechanism is provided for reducing the parameter space. Not only are the test parameters created in advance, but they are also coupled to the actual test code. Further, current test coverage tools do not track runtime configurations.

DESCRIPTION

This disclosure describes techniques to validate test coverage of software features under a matrix of runtime configurations. The techniques shift the focus of a feature developer or tester from keeping track of all production configuration variations to creating rules that describe the dimensions and configurations under which tests need to pass. Integration tests are monitored

and validated such that the rules defined for the feature are successfully tested against the matrix of feature combinations.

The techniques enable early and comprehensive test coverage for all relevant combinations across the matrix of features. Product teams can easily define use cases and feature requirements. Developers or testers can specify the dimension values a product needs to be tested against. They can track the status of testing against those dimensions for a given product version before the feature rolls out to production. The combination of compatibility expectations is automatically generated and integrated with test execution tooling to effect test coverage at the appropriate environment level. Checks can be automated and qualification guardrails provided to prevent untested features from reaching production.

The techniques can be implemented in the form of a test coverage tool, some features of which include:

- **Dimension specification**, which enables the developer/tester to specify the attributes and attribute values to be tested, the minimal subset needed to validate a feature, rules to reduce the cross-combination cardinality, simplifications such as equivalency classes to reduce the number of test runs, etc. Dimension values and test validation status are automatically kept up to date.
- **Automatic test monitoring**, which is integrated with a test execution workflow such that the values being tested, and pass/fail statuses are automatically recorded.
- **Coverage tracking**, which enables a developer/tester to quickly see the extent of test coverage and the presence of coverage gaps for a feature. Status reports include information such as how many dimensions have passed, how many are being tested but failing, how many haven't been tested, etc. Coverage tracking can illustrate the difference

between actual test coverage and the space of possible combinations of attribute values. Coverage gaps thus identified can be addressed by newly designed and targeted tests.

- **Release blocking**, a feature that can use the status report, including the presence or absence of coverage gaps, to generate an automated go/no-go signal before a feature is rolled out to production.
- **Rules specification**, which enables developers or testers to program configurable coverage requirements or expectations without knowing individual values and combinations. The requirements specified by the developer/tester are automatically mapped to actual test coverage.

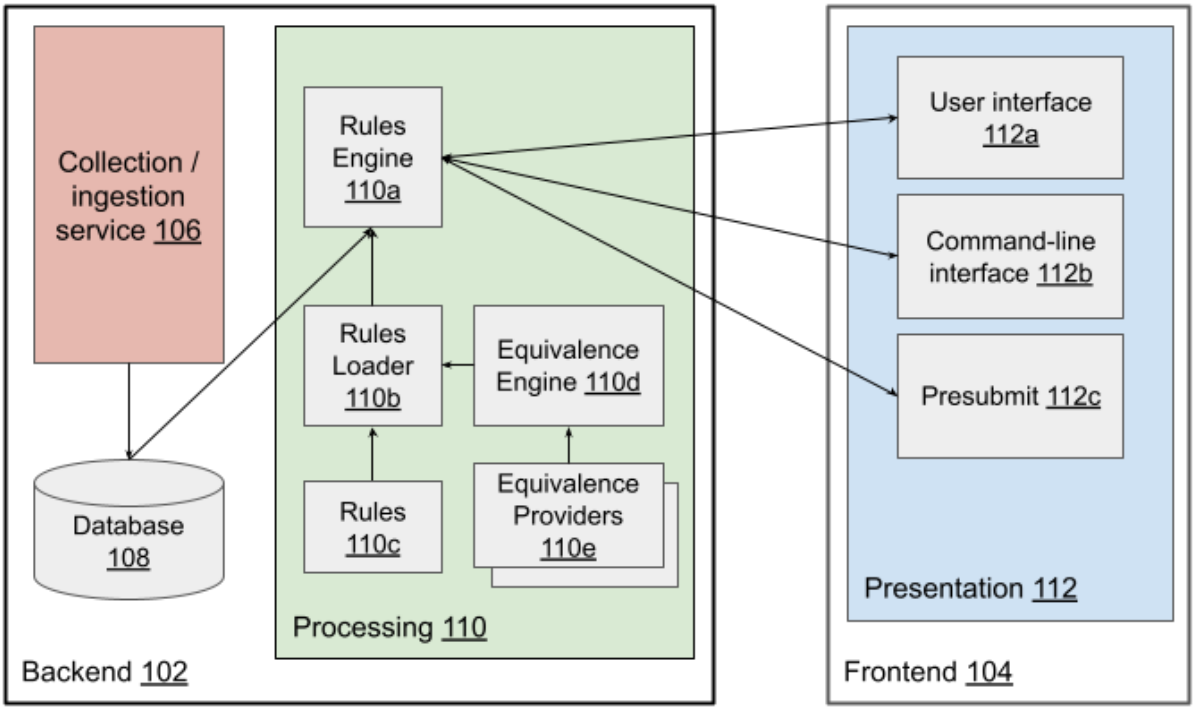

Fig. 1: Components of a tool to validate software functionality across combinations of runtime configurations

Fig. 1 illustrates the components of a tool to validate software functionality across combinations of runtime configurations. The tool is divided into two broad sections - a backend

(102) and a frontend (104). The backend comprises a data collection/ingestion service (106), a database (108), and a processing module (110). The collection/ingestion service gathers data such as the values of each attribute and historical test results from text execution workflows (or tools). The collection/ingestion service feeds such data into the database. The database schema supports adding new dimensions, dimension values changing over time, and correlating test coverage to a specific test target.

The processing module includes a rules engine (110a), which is fed rules (110c) by a rules loader (110b). Testing rules can be specified by the developer/tester. Rules can be specified that define important feature sub-spaces, e.g., feature subspaces to test in great detail; feature subspaces of relatively less importance; feature subspaces that can be ignored; etc. An example pseudocode for a rule is shown in Fig. 2.

The rules engine accesses the database and ensures that features are tested in accordance with the rules as specified by the developer/tester. The rules engine generates the attribute-value combinations per the rules, maps the required coverage to the actual coverage, and generates the coverage gap report. The rules loader can be fed equivalence classes by an equivalency engine (110d) operating with data provided by equivalency providers (110e). An equivalence class is a class of dimension values with no functional differences between class members, such that a pass (or fail) on one class member can be considered as a pass (or fail) on other class members. Equivalency classes can be used to reduce the cardinality of a test.

The frontend includes a presentation layer (112), which presents test coverage results in a user interface (112a), e.g., a reporting dashboard; enables, via a command-line interface (112b), user interaction with qualification procedures, coverage gap reports, and the data underlying the coverage results; enables, via a pre-submit module (112c), developers/testers to block releases

with coverage gaps from reaching customers; etc. In this manner, test data, developer-specified rules, and user requests are unified to enable a decision to be made on test coverage and to execute a go/no-go decision on rollout.

```
ruleset = {
 name = "My Feature"

 rules = [{
     name = "full validation"
     dimensions = [{
       type = machine_type
       values = ‘linux’
     }, {
       type = image
       values = ‘freeLinux101’
     }]
   }
 ]
}
```

Fig. 2: An example pseudocode test coverage rule

Fig. 2 illustrates an example pseudocode test coverage rule that can be programmed by the developer/tester, and which is used by the rules engine to determine gaps in test coverage. Rules define Boolean logic to validate the test coverage for one or more dimensions along with any required metadata. For rules incorporating a single dimension, coverage data is checked for that specific dimension; for rules that incorporate multiple dimensions, the cartesian product of the dimensions is checked.

As illustrated, a test coverage rule set can cover a number of features (‘My Feature’). A feature can be covered by a number of rules. A rule is defined by a name (‘full validation’), dimensions, values, etc. Values within a rule are similar to an allowlist. Each rule checks for

certain combinations of values across dimensions. Rules are managed independently of the tests under execution.

In this manner, the described techniques can rapidly determine test coverage gaps in multidimensional configuration spaces of substantial cardinality by:

1. enabling a developer/tester to define a software feature and its associated tests;
2. enabling a developer/tester to specify a record of configuration attributes (dimensions), with discrete, finite values;
3. enabling a developer/tester to define a set of rules associated with a feature and by forming a Boolean algebra over the coverage matrix that defines the required test coverage;
4. automatically monitoring tests such that the values being tested, and pass/fail statuses are automatically recorded;
5. reading zero or more values for each dimension from systems under test and adding them to the coverage matrix that describes the presence or absence of each value;
6. tracking the coverage of tests; and
7. automatically blocking a release if the test coverage gap is beyond a threshold.

Advantages of the described techniques include:

- The set of rules for a feature defines its test plan. The rules are decoupled from actual tests and can be maintained independently.
- The values for each dimension are decoupled from the same and can be maintained independently. This enables the required coverage to be kept up to date without affecting either the test plan or the test code.

- The rules enable irrelevant combinations to be ignored, both by defining a subset of required values and by allowing one value to substitute for another.
- Heuristics and additional data can be incorporated to intelligently reduce the combinations under validation.

The described techniques apply to any software that is to be tested against multiple independently controllable configurations. By establishing a canonical matrix for proving product functionality across configurations, untested combinations of configurations can be detected. Given a set of features and expectations for a product, the canonical matrix can enable the creation of individual compatibility tests needed to verify functionality under all configurations the product is expected to support. By removing the need to manually define configuration tests for a product and by enabling features to be developed independently of the products that use them, the described techniques enable product testing at scale. By dynamically maintaining the variations under test, intelligently prioritization of coverage becomes possible in a number of ways such as taking into account actual product usage; prioritizing coverage for new/risky features; developing equivalence classes to obtain coverage from related configurations; etc. Software testability is sustained in the face of exponential dimensional growth.

CONCLUSION

This disclosure describes techniques to determine test coverage gaps in multidimensional configuration spaces of substantial cardinality by enabling a software developer to define a set of rules associated with a feature and by forming a Boolean algebra over a coverage matrix that defines the required test coverage. Test coverage is tracked and presented on a dashboard. Release can be blocked automatically if the test coverage gap is beyond a threshold. Based on

the coverage matrix, untested combinations of configurations can be detected and individual compatibility tests to verify functionality can be designed. By dynamically maintaining the allowed values of the configuration space, intelligent prioritization of coverage becomes possible. Software testability is sustained in the face of exponential dimensional growth.