Pseudo-Exhaustive Verification (PEV)
User Guide

The Pseudo-Exhaustive Verification (PEV) software is designed to make it easy to test rule-based systems, such as access control, rule-based expert systems, or business rule engines. Key features:

- Automated test generation – the user specifies rules, then the tool will automatically generate test conditions for the rules
- Easy to learn and use – worked example included in this guide
- Produces tests for both positive *and* negative conditions (usually the hard part of testing), covering *all t*-way combinations of parameter values, complete up to a designated level of *t* (typically 2 to 4-way).
- Only a few minutes per rule to enter rule, then generate and save tests, generally much faster than conventional approaches to testing these systems
- Produces test values in comma-separated value format, for input to a spreadsheet or to other tools used in mapping test values to the input format required by the system under test
- Usable on any platform that supports the Java runtime library

PEV was developed by the National Institute of Standards and Technology, Information Technology Laboratory, Computer Security Division (NIST/ITL/CSD). To ensure portability across all common platforms, the software is packaged as a Java Archive (JAR) file which will run on any system supporting Java, and is directly executable as a Graphical User Interface (GUI). It can also be run as a Command Line Interface (CLI) from a terminal, which may be useful for incorporating PEV into a tool chain for test automation. The PEV tool can be downloaded freely from:
https://github.com/usnistgov/combinatorial-testing-tools/blob/master/pev.jar

The PEV software has been designed to accept Boolean rules or policies comprised of Boolean variables, Boolean operators and relational expressions. The software will parse the Boolean policy, then generate tests that cover all positive and negative conditions, up to a specified level of t-way interaction.

1 Example and Overview

The PEV testing process will be introduced with an example that will also be used in the remainder of the document, which explains tool functions.

Suppose we wish to test a set of business rules that are used in determining employee access to various positions in the organization. One such rule is the following:

> rule: "If the individual is an employee and is age 18 or older and either has first aid training, or an EMT certification, or a medical degree, then authorize"

The first step is to determine the attributes or parameters that will be used in testing. In this case these are:

> attributes: *employee* , *age*, *first_aid_training*, *EMT_cert*, *med_degree*

The rule can then be stated in the logic notation that is used by the PEV tool, with suitable abbreviations to make rule entry easier:

rule: *emp* && *age* > *18* && *(fa* || *emt* || *med)* → *grant*
else → *deny*

At this point we can begin entering the rule into the PEV tool, then generate tests in a .csv file format.

2 Combinatorial Testing

This section is not meant to be a comprehensive discussion on the topic; rather a brief primer on the subject. See Ref. [1] for a more thorough explanation of this method. Combinatorial testing is a method for testing software where all *t*-way combinations of input parameters are tested; the resulting test vectors are a much smaller subset. For example, given 10 Boolean variables (A to J) there would be 1,024 possible inputs exhaustive testing. Using combinatorial methods, it only takes 13 tests to cover all 3-way combinations. Empirical data show that testing all 3-way combinations will generally detect 80-95% of errors, and covering all 5-way to 6-way combinations is sufficient for nearly 100% fault detection. Because of this potential to provide the same fault detection as exhaustive testing, combinatorial methods are sometimes referred to as pseudo-exhaustive or effectively-exhaustive.

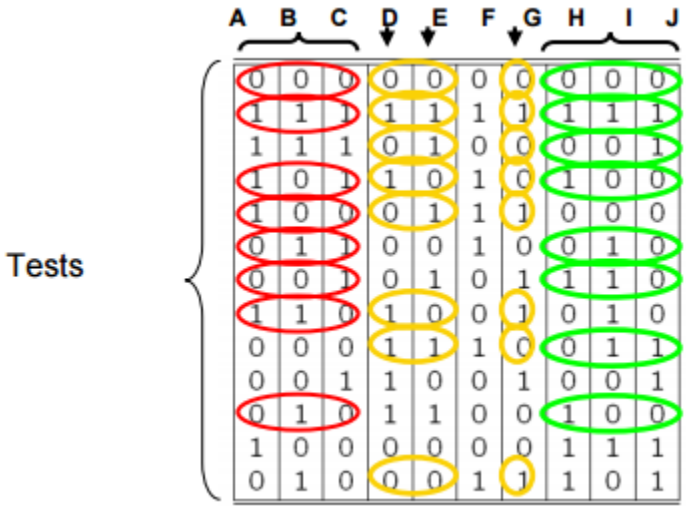


Figure 1 - 3-way combinations of 10 Boolean Variables

When looking at any 3 columns all possible combinations of the 3 Boolean values can be found {000, 001, 010, 011, 100, 101, 110, 111}.

3 Software Details

3.1 Software Execution

The PEV software can be run as two different interfaces – GUI and CLI.

Most operating systems will support running the JAR file by double clicking the icon. If not – the GUI can be executed by running the terminal command:

```
java -jar pev.jar
```

To run the CLI, parameters must be passed through the terminal command indicating that CLI is the desired interface to run:

```
java -jar pev.jar -cli -policy="<Input Policy>" -var="variableName(minimum_value,
maximum_value)" [multiple variables] ... -outpath="<Output Path>"
```

An example of running PEV in CLI mode:

```
java -jar pev.jar -cli -policy="emp & age > 18; & (fa | emt | med) | b < 3;" -
var="age(0,22)" -var="b(0,13)" -outpath="C:\Users\Public\Desktop\pev_output.csv"
```

3.2 Parsing

Parsing is a critical step of the PEV software, and occurs before any testing is performed. Since the software needs to accept input from the user, any input must be modified and sanitized prior to use. The parser will strip extraneous whitespace, and then normalize disparate Boolean operations `(&&, &, ||, |, !, ~)`. The parser will attempt to match open and closing parenthesis. This sanitization is to ensure compatibility with the various APIs used throughout the software, as well as to catch any syntactical problems prior to testing.

Additionally, the software has initial support for relational expressions (e.g., b < 3;). However, the PEV software tests a Boolean policy, it will treat "b < 3;" as a Boolean; PEV can do this because there are values where it passes and values where it fails. During parsing, PEV will locate numeric relational expressions and replace them with temporary Boolean variables. After the replacement, the policy is processed as normal. The relational values are solved at a later step and the results are recorded.

Table 1 - Supported Operators

Name	Operator	Normalized Operator
AND	&&, &	&
OR	||, |	|
NOT	!, ~	!
GREATER THAN	>	
GREATER THAN OR EQUAL	>=	
LESS THAN	<	
LESS THAN OR EQUAL	<=	
EQUAL TO	=	
NOT EQUAL TO	!=	

Once the initial input policy is parsed, the software will convert it to Disjunctive Normal Form (DNF) to be tested. The user will be presented with a breakdown of the DNF policy (split on the OR statements), each part is a grant condition which needs to be solved. Additionally, the user can set minimum and maximum values for any relational variable found in the policy.

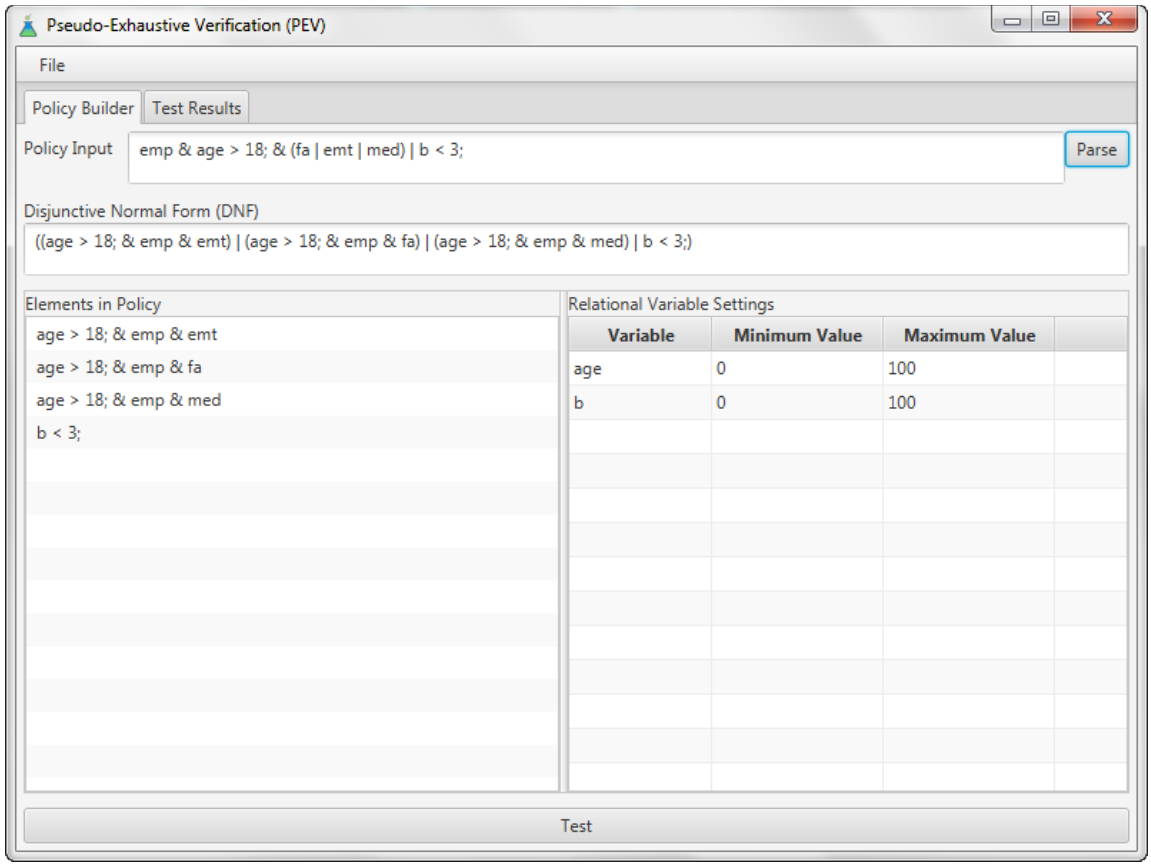


Figure 2 - PEV software, after initial policy has been parsed

3.3 Solve for Grant Conditions

Each individual expression between OR operators is an expression that, once solved, will produce one grant condition. These expressions represent the only possible grant conditions for the original policy – so it is possible to produce exhaustive grant conditions.

In Figure 2 above, the original input policy:

```
emp & age > 18; & (fa | emt | med) | b < 3;
```

→ Converted to DNF →

```
((age > 18; & emp & emt) | (age > 18; & emp & fa) | (age > 18; & emp & med) | b < 3;)
```

Splitting on the OR operators, there are four individual expressions for the grant conditions (replacing relational expressions with temporary Boolean variables `tmp0 = age > 18;` and `tmp1 = b < 3;`):

- `tmp0 & emp & emt`
- `tmp0 & emp & fa`
- `tmp0 & emp & med`
- `tmp1`

To solve these expressions, any variable present is evaluated with the following rules:

1. **Non-negated variables evaluate to true**

2. Negated variables evaluate to false
3. *Variables not present evaluate to false*

Table 2 - Solved Grant Conditions

Expression	tmp0	tmp1	emp	emt	fa	med
tmp0 & emp & emt	**True**	*False*	**True**	**True**	*False*	*False*
tmp0 & emp & fa	**True**	*False*	**True**	*False*	**True**	*False*
tmp0 & emp & med	**True**	*False*	**True**	*False*	*False*	**True**
tmp1	*False*	**True**	*False*	*False*	*False*	*False*

3.4 Solve for Deny Conditions

Depending on the complexity of the input policy, it may not be feasible to produce exhaustive deny condition output combinations. By utilizing combinatorial test methods, it is possible to generate covering arrays of sufficient strength to have good test coverage. The method for producing deny conditions can be found by generating the full covering array for all the unique Boolean variables within the policy, and using the DNF policy as a constraint – which will remove the grant conditions from the resulting output

To perform this task, PEV utilizes NIST's Automated Combinatorial Testing for Software (ACTS). The PEV creates an internal instance of the ACTS software, and passes a list of the unique Boolean variables from the policy (including temporary Boolean replacements for relational expressions). The next step is to add the DNF policy as a constraint to the system – so that the grant conditions are not included as deny results. Finally, the n-way combination is dynamically set (between 2 and 6) based upon the longest combination of Boolean variables ANDed together – in this example, the value of 3 is set.

Table 3 - Solved Deny Conditions

tmp0	tmp1	emp	emt	fa	med
True	False	True	False	False	False
True	False	False	True	True	True
False	False	True	True	True	False
False	False	False	False	False	True
False	False	False	True	False	False
False	False	True	False	True	True
True	False	False	False	True	False
False	False	True	True	False	True
True	False	False	True	False	True
False	False	False	False	True	False
True	False	False	False	False	True
True	False	False	True	False	False

3.5 Solve for Relational Expressions

The final processing step is to solve the relational expressions; the Choco Expression Parser is used, which itself utilizes the Choco Constraint Solver.

Relational Expression Formatting

To utilize the relational expressions, a specific format must be adhered to (currently, only integers can be tested). The general format is:

```
Variable OPERATOR Integer_Value;  Or  Integer_Value OPERATOR Variable;
```

Every relational expression must end with a semicolon (;), and two or more relational expressions in a row (without Boolean operators between them) will be replaced with one temporary Boolean variable during parsing. For example:

Table 4 - Input Policies and Resulting Parsed Policy

Input Policy	Parsed Policy
`a > 10; 20 < b; || n`	`tmp0 || n`
`a > 10; || 20 < b; || n`	`tmp0 || tmp1 || n`

After being extracted and replaced by temporary Boolean variables, and the Grant/Deny conditions are found, an instance of Choco Expression Parser is created, and the relational expressions are passed as parameters. The minimum and maximum range for the expression to test against must be set – the PEV GUI includes a section which will allow the adjustment of every relational variable minimum and maximum values (default set to 0 to 100). These values can be adjusted prior to testing the policy so that a customized range can be found.

The solutions to the solved expressions are then placed into the results where appropriate.

3.6 Results

Once testing is completed, the PEV software will display some usage metrics (Fig. 3) and then the parameters which will result in grant conditions, as well as the covering array for deny conditions (Fig. 4). At this point, the results can be saved as a comma separated value (CSV) file to be used for testing, by clicking File→Save Result (Fig. 4).

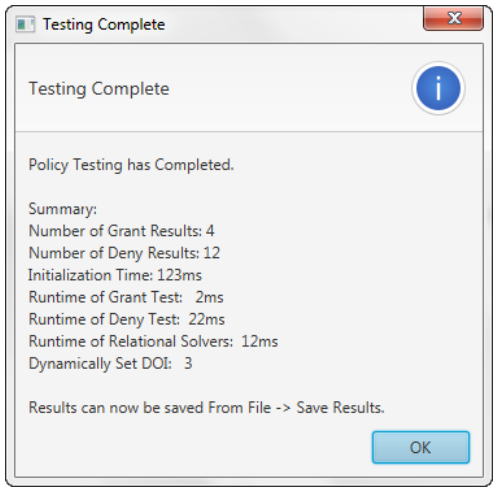


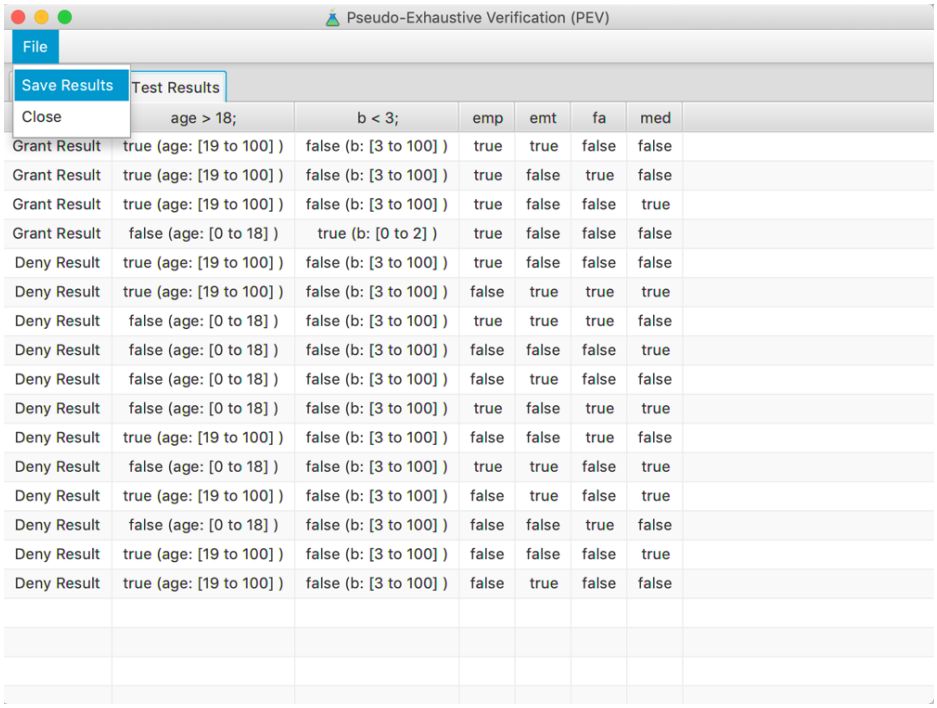
Figure 3 – PEV Test Summary

Notice that the relational expressions (e.g., age >18) have been instantiated with value ranges that make them either true or false. For testing, these ranges can be replaced with random values between the endpoints, or by selecting one of the endpoints in the .csv file. For example, a spreadsheet could be used to read the .csv file, then replace all but the desired number; in Excel, to keep the lower bound the expressions would be

find: true*[replace:

find: to*) replace:

That is, we replace the designated find strings with null or space, leaving the lower bound (it is important that the find and replace be done by column, to avoid deleting values in more than one parameter spec at a time).



	age > 18;	b < 3;	emp	emt	fa	med
Grant Result	true (age: [19 to 100])	false (b: [3 to 100])	true	true	false	false
Grant Result	true (age: [19 to 100])	false (b: [3 to 100])	true	false	true	false
Grant Result	true (age: [19 to 100])	false (b: [3 to 100])	true	false	false	true
Grant Result	false (age: [0 to 18])	true (b: [0 to 2])	true	false	false	false
Deny Result	true (age: [19 to 100])	false (b: [3 to 100])	true	false	false	false
Deny Result	true (age: [19 to 100])	false (b: [3 to 100])	false	true	true	true
Deny Result	false (age: [0 to 18])	false (b: [3 to 100])	true	true	true	false
Deny Result	false (age: [0 to 18])	false (b: [3 to 100])	false	false	false	true
Deny Result	false (age: [0 to 18])	false (b: [3 to 100])	false	true	false	false
Deny Result	false (age: [0 to 18])	false (b: [3 to 100])	true	false	true	true
Deny Result	true (age: [19 to 100])	false (b: [3 to 100])	false	false	true	false
Deny Result	false (age: [0 to 18])	false (b: [3 to 100])	true	true	false	true
Deny Result	true (age: [19 to 100])	false (b: [3 to 100])	false	true	false	true
Deny Result	false (age: [0 to 18])	false (b: [3 to 100])	false	false	true	false
Deny Result	true (age: [19 to 100])	false (b: [3 to 100])	false	false	false	true
Deny Result	true (age: [19 to 100])	false (b: [3 to 100])	false	true	false	false

Figure 4 – To save results in a .csv file, click on File → Save Results

Converting the output in the .csv files will necessarily involve user-supplied scripts or other tools to produce the final output that will be used in calling and executing the system under test.

It is also possible to run PEV in a command line mode, which may be convenient when using the PEV tool from shell scripts. Fig. 5 illustrates the use of PEV from a command line interface.

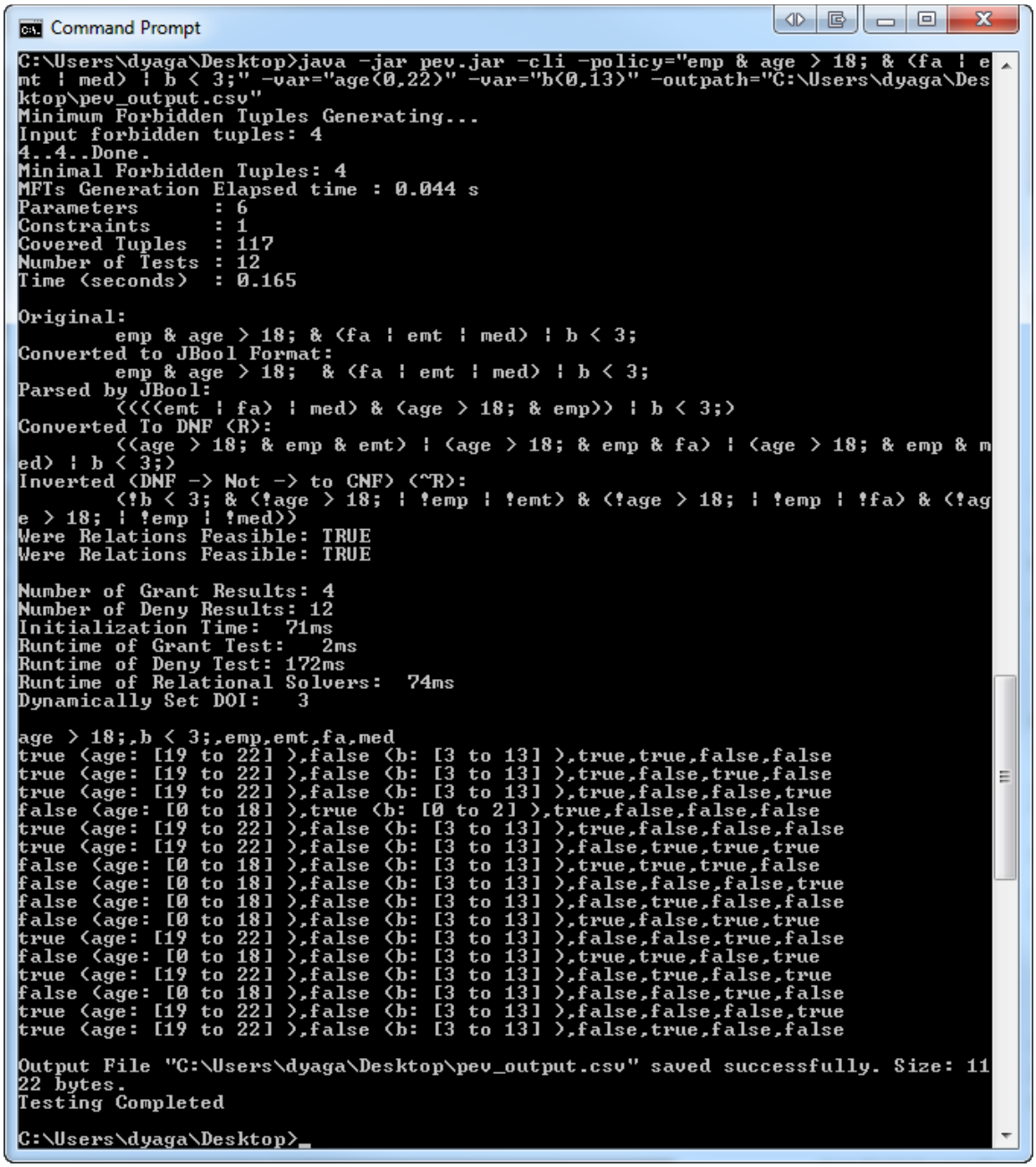


Figure 4 - PEV in Command Line Interface

4 External JAVA Libraries

The software was developed using the JAVA programming language. It utilizes several existing JAVA Archive (JAR) libraries to perform the testing.

- To convert the input Boolean policy to DNF: **jbool_expressions**
  - https://github.com/bpodgursky/jbool_expressions
- To solve relational statements: **choco-exppar** and **choco-solver**
  - https://github.com/kaktus40/choco-exppar
  - http://www.choco-solver.org/
- To generate covering arrays: **NIST ACTS**
  - http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html

5 References

[1] *Practical Combinatorial Testing*, D. Richard Kuhn, Raghu N. Kacker, Yu Lei https://dx.doi.org/10.6028%2FNIST.SP.800-142

[2] Kuhn, R., Yaga, D., Kacker, R., Lei, J., & Hu, V. C. (2018, July). Pseudo-Exhaustive Verification of Rule Based Systems. In *SEKE* (pp. 586-585).