Input Space Coverage Matters

D. Richard Kuhn, NIST, kuhn@nist.gov
Raghu N. Kacker, NIST, raghu.kacker@nist.gov
Yu Lei, University of Texas at Arlington, ylei@uta.edu
Dimitris Simos, SBA Research, DSimos@sba-research.org

Abstract. Testing is the most commonly used approach for software assurance, yet it remains as much judgement and art as science. Structural coverage adds some rigor to the process by establishing formally defined criteria for some notion of test completeness, but even full coverage, however defined, may miss faults related to rare inputs that were not included in the test suite. We suggest that structural coverage measures must be supplemented with measures of input space coverage. Useful input space measures exist and have a relationship with structural coverage measures, providing a means of verifying that an adequate input model has been defined.

Introduction

One of the key objections to testing as a form of software verification is that it is never possible to show that the system under test works for all possible inputs. It is also difficult to provide meaningful statements about the adequacy of a test set for verifying that the system under test (SUT) works correctly. Conventional structural coverage measures, typically statement or branch coverage, leave much to be desired. Even if all statements are executed and all branches are taken, there is no guarantee that the input space has been covered adequately for fault detection. A latent error may show up later with the appearance of a very rare combination of conditions that was not included in testing. Methods of systematically partitioning the input space have been studied extensively, but most necessarily involve a good deal of subjective judgement, and do not provide quantitative measures of completeness. Combinatorial methods offer ways to build on existing techniques for input space partitioning, to provide more rigorous testing.

Coverage Measures

A complete input model is part of the goal of achieving thorough testing. Depending on what system aspects are to be considered in defining completeness, a variety of approaches exist to determining when testing is considered enough. Typically, these include some notion of fully covering requirements, and may also consider structural coverage of the code. In software engineering, *structural coverage* refers to measures of the degree to which programs have been exercised. Two of the most widely used measures are statement coverage, the proportion of program statements that have been executed, and branch coverage (also known as decision coverage), the proportion of branches that have been evaluated to both *true* and *false*. Many other measures or test criteria exist, including condition coverage and modified condition decision coverage, and it can be shown that these criteria form a hierarchy [4]. For example, decision coverage subsumes statement coverage. Structural coverage measures are of value in gauging the thoroughness of a test set, although their utility is somewhat limited. Statement coverage is the weakest of these measures, but failure to achieve full statement coverage at least indicates that code has not been tested well enough. Branch coverage provides a stronger measure, and more complex criteria related to branch coverage are used for some life critical applications such as aviation.

Of course, simply achieving some level of structural coverage is hardly adequate, because it must be shown that the SUT accomplishes its intended function. If the code does only one of ten required functions, the level of structural coverage is obviously an inadequate measure for determining if all of the ten functions have been implemented. The SUT may work well for the subset of inputs related to the implemented function, but it would fail if presented with inputs directing execution of unimplemented functions. Similarly, if all of the functions are implemented but not all correct, the code may produce the right result for some input values but fail on others.

Among the key questions in software testing and assurance are what parameters matter, and what values should be included in testing. Except for trivial problems, it is generally intractable to test all possible inputs, so some form of equivalence partitioning must be used, i.e., the input space is divided into sets of inputs that are considered equivalent with respect to some relation that is meaningful for the software under test. Put simply, the problem is to find parameter values that each adequately represent a much larger set of values. A simple example is partitioning into valid and invalid input values. Another example might be ranges of weights or sizes that are equivalent with respect to the shipping charge for each class.

Because the input space is far too large to test exhaustively, inputs must be discretized for continuous-valued variables, or a small subset selected from enumerated values. This problem is fundamental in software assurance, and an extensive body of research has been developed to solve it. In general, input parameters cannot be considered in isolation, because the value of one may limit the values that must be considered in testing for other parameters, that is, there may be constraints among parameters.

How can an adequate input model be found? Part of the problem was solved decades ago, with systematic methods of analyzing and partitioning the input space. Accepted practices for this task include boundary value analysis [5,4], to identify boundaries in range-defined variables and divide the input space into partitions of values for which the system under test can be expected to produce equivalent results. The classic category partition method [1] provides a systematic way to integrate the definition of boundary values and equivalence partitions into test frames that consolidate values into sets that exercise particular functionality in the SUT. A more recently developed approach that builds on these ideas is the classification tree method [2], which adds a graphical notation and analysis of hierarchical or implicit dependencies among parameter values. Such methods are valuable in providing a systematic, rigorous method of input parameter model definition, but a good deal of engineering judgment is still required to determine whether the model is adequate. This partition of the parameters and values is referred to as an input parameter model (IPM), or simply input model.

After an input model has been constructed, how can one ensure that it is adequate for testing, that is, the equivalence partitions contain values that are truly equivalent in terms of system response? More generally, can we find measures of the IPM that are relevant to testing? To answer this question, it is useful to first consider empirical data on the distribution of faults.

Background – Fault Distribution.
Empirical data show that most failures are triggered by a single parameter value, or interactions between a small number of parameters, generally two to six [ref], a relationship known as the *interaction rule*. An example of a single-value fault might be a buffer overflow that occurs when the length of an input string exceeds a particular limit. Only a single condition must be true to trigger the fault: *input length* > *buffer size*. A 2-way fault is more complex, because two particular input values are needed to trigger the fault. One example is a search/replace function that only fails if both the search string and the replacement string are single characters. If one of the strings is longer than one character, the code does not fail, thus we refer to this as a 2-way fault. More generally, a *t*-way fault involves *t* such conditions.

Figure 1 shows the cumulative percentage of faults (y axis) at $t = 1$ to 6 (x axis) for various real applications [8,15]. We refer to the distribution of *t*-way faults as the *fault profile.* Figure 1 shows the fault profile for a variety of fielded products in different application domains, and results for initial testing of a NASA database system. As shown in Figure 1, the fault detection rate increases rapidly with interaction strength, up to t=4. With the medical device applications, for example, 66% of the failures were triggered by only a single parameter value, 97% by single values or 2-way combinations, and 99% by single values, 2-way, or 3-way combinations. The detection rate curves for the other applications studied are similar, reaching 100% detection with 4 to 6-way interactions.

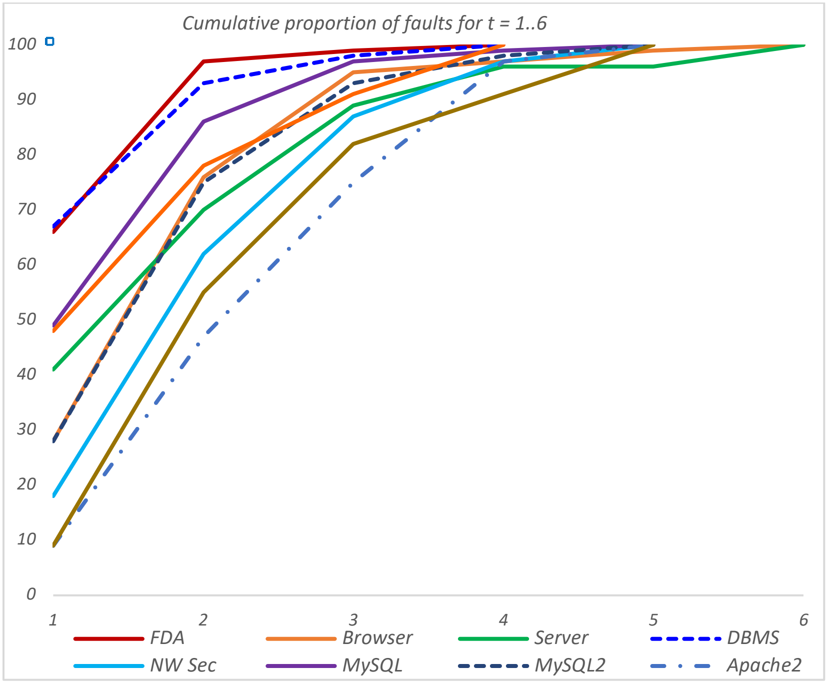

Figure 1. Cumulative fault distribution

Matrices known as *covering arrays* can be computed to cover all *t*-way combinations of variable values, up to a specified level of *t* (typically $t \leq 6$), making it possible to efficiently test all such *t*-way interactions [7]. The effectiveness of any software testing technique depends on whether test settings corresponding to the actual defects in the SUT are included in the test sets. When test sets do not include test settings corresponding to actual defects, the faults and defect will not be detected. Conversely, with the right input model, we have evidence that the software works correctly for *t*-way combinations contained in passing tests.

Coverage Implications of Fault Distribution

The empirical distribution of faults suggests that it is useful to consider the degree to which combinations of input values are covered in a test set. The data in Fig. 1 suggest that we may miss 5-10% of possible errors if we do not test 4-way, or more complex, combinations. However, these findings also mean that in general it will not be necessary to test all possible combinations of input values, because relatively few factors are involved in failures. Covering *t*-way combinations of inputs, for small values of *t*, is to some extent equivalent or at least close to exhaustive testing. Thus, it is important to understand the coverage of input combinations for any test set.

As noted, a covering array may be constructed to cover all *t*-way combinations of input parameters, but any test set of course contains many combinations. We can measure the *combinatorial coverage*, i.e., the coverage of *t*-way combinations in a test set, for a better understanding of test set quality. These measures provide quantitative levels of quality very different from conventional structural coverage. In particular, combinatorial coverage has a direct relationship with fault detection. As shown in Fig. 1, a significant portion of fault triggering combinations involve more than two factors, so the level of combination coverage measures the ability of the test set to detect, for example, faults induced by 4-way or 5-way combinations. These measures can also be computed independently of structural coverage, prior to running any tests, because they relate to the (static) content of the test set. However, there is an interesting relationship between combinatorial coverage and structural coverage, as discussed later.

Measuring Coverage of Fault-triggering Combinations

Measuring combination coverage can help in understanding the degree of risk that remains after testing. If a high level of coverage of input state-space variable combinations has been achieved, then the risk is small that there is latent combination that may induce a failure. Lower coverage reflects greater risk that there is some untested failure-triggering combination. A variety of measures of combinatorial coverage can help in estimating this risk. In this section we describe some of the basics; see, [15] and [18] for more details.

For a set of *t* variables, a *variable-value configuration* is a set of *t* valid values, one for each of the variables, i.e., the variable-value configuration is a particular setting of the variables. For example, four binary variables *a, b, c*, and *d*, for a selection of three variables *a, c*, and *d* the set {a=0, c=1, d=0} is a variable-value configuration, and the set {a=1, c=1, d=0} is a different variable-value configuration.

a	b	c	d
0	0	0	0
0	1	1	0
1	0	0	1
0	1	1	1

Table 1. Test array with four binary components

Vars	Configurations covered	Config coverage
a b	00, 01, 10	.75
a c	00, 01, 10	.75
a d	00, 01, 11	.75
b c	00, 11	.50
b d	00, 01, 10, 11	1.0
c d	00, 01, 10, 11	1.0

Table 2. The test array covers all possible 2-way combinations of a, b, c, and d to different levels.

It is also useful to measure the number of t-way combinations covered out of all possible settings of t variables. For a given combination of t variables, *total variable-value configuration coverage* is the proportion of all t-way variable-value configurations that are covered by at least one test case in a test set. This measure may also be referred to as total t-way coverage. For the array in Table I, there are 4-choose-2, C(4,2) = 6 possible variable combinations and $2^2 \times C(4,2) = 24$ possible variable-value configurations. Of these, 19 variable-value configurations are covered and the only ones missing are ab=11, ac=11, ad=10, bc=01, bc=10, so the total variable-value configuration coverage is 19/24 = 79%. Although the example in Table 1 uses variables with the same number of values, it is also possible to compute coverage for test sets in which parameters have differing numbers of values.

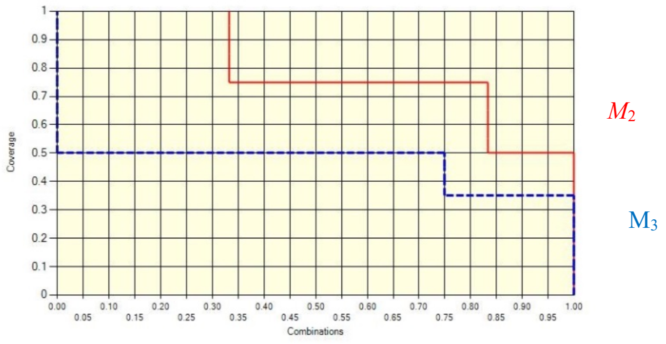

Figure 2. Graph of coverage for tests in Table 1

Figure 2 shows a graph of the 2-way (red/solid) and 3-way (blue/dashed) coverage data for the tests in Table 1. Combination coverage of variable values is given as the y axis, with the percentage of combinations reaching a particular coverage level as the x axis. Of particular interest is the minimum t-way coverage, M_t, which is the smallest proportion of coverage of variable-value configurations for parameters taken t at a time. For example, Table I shows four binary variables, a, b, c, and d, where each row represents a test, so parameters taken two at a time have four possible configurations: 00, 01, 10, 11. The six possible 2-way variable combinations, *ab, ac, ad, bc, bd, cd*, only *bd* and *cd* are covered to different levels, shown in the second column. The minimum coverage for 2-way combinations, $M_2 = .5$, shown as $y = .5$ at $x = .833$ because one out of the six variable combinations has 2 of the 4 possible settings of two binary variables covered. The area under the curve for 2-way combinations is approximately

79% of the total area of the graph, reflecting the total coverage of 19 out of 24 2-way combinations.

Practical Example

Combinatorial coverage measures were originally developed to analyze the input space coverage of tests for spacecraft control software [9][10]. A test suite of 7,489 tests had been developed using conventional techniques such as use case analysis and knowledge of likely error sources. The control system included 82 variables, with a test configuration of $1^3 2^{75} 4^2 6^2$ (three 1-value, 75 binary, two 4-value, and two 6-value). Figure 3 shows combinatorial coverage achieved by the full set of 7,489 tests (red = 2-way, blue = 3-way, green = 4-way, orange = 5-way). The area under the curves is the proportion of the input space tested at the various t-way levels, while the are above the curve represents untested space – where errors might still be found. For example, 2-way coverage is 94%, so there is relatively little risk of faults that might be triggered by 2-way combinations.

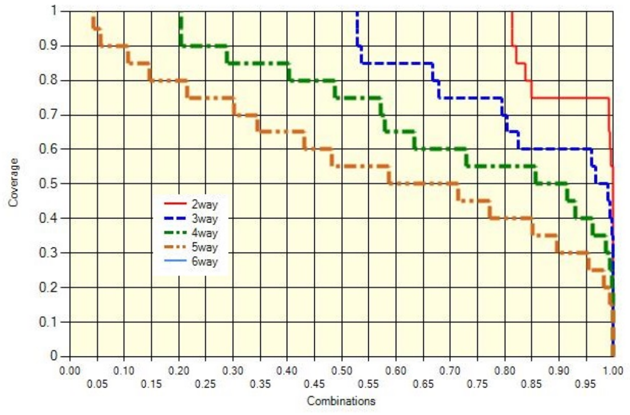

Figure 3. Configuration coverage for spacecraft example.

interaction	combinations	settings	coverage
2-way	3321	14761	94.0
3-way	88560	828135	83.1
4-way	1749060	34364130	68.8
5-way	27285336	603068813	53.6

Table 3. Total t-way coverage for Fig. 3 configuration.

Relationship Between Combinatorial Coverage and Structural Coverage

It is obvious that any thorough assessment of a system must show that all requirements have been met, but it is also true that the system should not produce unexpected behavior. One of the most effective means of confirming these objectives is to generate tests from requirements, then ensure that full structural coverage has been achieved, for some strong coverage criterion. That is, tests must be requirements-based; structural coverage is used only to validate the quality of these tests. This approach has been required by the US Federal Aviation Administration for testing life-critical aviation software, in RTCA standard DO-178B [13], and continued in the update DO-178C [14]. This requirement uses a stronger relative of branch/decision coverage called modified condition decision coverage (MCDC), which subsumes branch coverage. (Branch coverage requires that each branch of every control structure, such as if or while conditionals, has been taken in the test suite. MCDC requires that every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to independently affect the decision outcome, and that each entry and exit point have been invoked at least once.). The intuition is clear – if full structural coverage has not been achieved, then either tests are inadequate, or there is some system functionality that is outside of the specification.

Structural coverage is a useful heuristic for evaluating whether we have a correct input model. FAA/Chilenski [11] notes that "MCDC should be used as a structural coverage criterion because it attempts to provide a cost effective form of logic verification. In essence, MCDC can be thought of as a (weak) measure of the coverage of equivalence classes and boundary values." According to DO-178B [12],

> If branches (decision outcomes) are left uncovered by the requirements-based verification process, then that process failed to consider the special cases the development process thought were important. Decision coverage ensures that the verification process has considered sufficient operational scenarios to execute every special case the system was designed for in at least one operational context.

One interesting development from combinatorial testing research is a result that shows a connection between (static) combinatorial coverage of the input space, and (dynamic) structural coverage [16]. This connection provides a formal justification for the DO-178B requirement that modified condition decision coverage (and consequently branch coverage) be achieved as a way to verify the adequacy of equivalence classes and boundary values.

As noted, structural coverage is a dynamic measure in the sense that it requires code execution. Combinatorial coverage, as we have defined here, looks at the input space, a static measure of value combinations. Clearly, the input to a program must affect the paths executed in the code, and thus impact structural coverage measures. In fact, we can show that there is a relationship between the coverage measures introduced above, and structural coverage, captured in simple theorem defining what we call the *branch coverage condition* [16].

> *Branch coverage condition*: A test set provides 100 % branch coverage for *t*-way conditionals if $M_t + B_t > 1$, where M_t = minimum combinatorial coverage at level t, and B_t =minimum proportion of t-way combinations that is guaranteed to trigger a branch within the code, where all variables in decision predicates have values from the variable set with minimum coverage characteristic M_t.

The reason this condition is important is that when test sets are based on covering arrays, which by definition have M_t = 100%, for a t-way covering array, then branch coverage should be 100%. The term B_t refers to the minimum proportion of input variable-value combinations that trigger a branch. For example, in the statements `if (A&&B) line1; if (C) line2;`, where A and B are Boolean variables, B_t = .25, because line1 is executed only for one of the possible four settings of A&&B.

A corollary also shows that branch coverage condition is obtained with $M_t > 1 - \frac{k}{v^t}$, if *k* or more *t*-way settings satisfy every predicate where all variables in decision predicates have values *v* from the variable set with minimum coverage characteristic M_t. If one uses a covering array in testing, how can $M_t + B_t > 1$ ever be false? In fact it cannot, if the input space has been properly modeled, that is, if variable value partitions have been properly defined. We will refer to M_t as the minimum coverage of an adequate input model, and M_t' as the minimum coverage of the input model that has been produced. That is, M_t is for an ideal IPM that has not necessarily been produced, while M_t' is measured on the current IPM, which may need to be revised.

To clarify, if M_t' is from a covering array for an adequate IPM, then $M_t' + B_t > 1 \Rightarrow$ branch coverage = 1.0, so branch coverage < 1.0 $\Rightarrow M_t' + B_t \leq 1$, but $B_t \geq v^{-t}$ for all parameters and M_t' = 1.0 because it is from a *t*-way covering array. Therefore, M_t' is not from an adequate IPM and must be revised. This result gives us a formal basis for the DO-178B approach of using MCDC to validate a requirements-derived test set.

So, failure to achieve the branch coverage condition indicates that the input model must be revised. The corollary above shows it is possible to detect a deficient input model even if 100% branch coverage is not achieved, a result that can be useful in developing the IPM. Note however we do NOT want to imply that less than full structural coverage is adequate for testing, only for developing an adequate IPM. For the full testing process, incomplete structural coverage indicates a different aspect of deficient testing.

One study that illustrated the relationship between input models and coverage nicely is [17]. The initial input model definition produced branch coverage in the range of approximately 70-75%, but by redesigning the IPM, 100% branch coverage and MCDC coverage (subsumes branch coverage) was obtained. The branch coverage condition provides a theoretical justification for the heuristic of requiring structural coverage as a means of validating requirements-based tests.

Note that achieving 100% branch coverage with a particular level of *t* does not indicate that higher strength testing is not needed. For example, if we find full branch coverage with 2-way testing, it will still be important to cover higher interaction strengths, because some failures may involve multiple factors that depend on nested conditions or variable states. That is, full branch coverage can be considered as necessary, but not sufficient, condition for achieving an adequate IPM. Unavoidably, some engineering judgement will be involved in selecting representative values for the model but measuring branch or MCDC coverage provides an assured method of validating IPM adequacy. Conventional methods of input space partitioning, such as the category partition and classification tree methods [refs], are highly effective for this. Combinatorial coverage measures described here can be used to improve the test engineering discipline, and provide measures of risk related to the untested input space.

Conclusions

Despite its shortcomings, testing is still the most widely used method of software assurance, but better measures are needed for quality of the test design. The most common measures of test completeness are various forms of structural coverage, but more can be done to improve the test engineering process. Structural coverage is a dynamic measure that should be supplemented with static measures of the input space. Combinatorial coverage measures the proportion of the input space relevant to testing that is covered by tests. Because only a small number of variables are involved in failures, measuring the proportion of *t*-way combinations covered in testing provides information on residual risk when the software is released.

This approach of using both static and dynamic measures also provides a means of ensuring that the input space partitioning has been done correctly. A simple theorem defining the branch coverage condition shows that if full branch coverage is not achieved when input space coverage is sufficiently high, then the input model has not been defined correctly, and must be revised. Tools exist to compute the necessary measures, and can be a valuable addition to the test engineering process [15][18].

Acknowledgements. This work was performed partly under the following financial assistance award 70NANB18H207 from U.S. Department of Commerce, National Institute of Standards and Technology.

References

1. Ostrand, T. J., & Balcer, M. J. (1988). The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, *31*(6), 676-686.
2. Grochtmann, M., & Grimm, K. (1993). Classification trees for partition testing. *Software Testing, Verification and Reliability*, *3*(2), 63-82.
3. Richardson, D. J., & Clarke, L. A. (1985). Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, (12), 1477-1490.
4. Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.
5. Beizer, B. (2003). *Software testing techniques*. Dreamtech Press.
6. Weyuker, E. J., & Ostrand, T. J. (1980). Theories of program testing and the application of revealing subdomains. *IEEE Transactions on software engineering*, (3), 236-246.
7. D.R. Kuhn, D.R. Wallace, A.J. Gallo, Jr., "Software Fault Interactions and Implications for Software Testing", IEEE Trans. on Software Engineering, vol. 30, no. 6, June, 2004.
8. NIST Special Publication 800-142*, Practical Combinatorial Testing*, Oct. 2010.
9. Maximoff, J. R., Kuhn, D. R., Trela, M. D., & Kacker, R. (2010, April). A method for analyzing system state-space coverage within a t-wise testing framework. In *2010 IEEE International Systems Conference* (pp. 598-603). IEEE.
10. Kuhn, D. R., Kacker, R. N., & Lei, Y. (2015). Combinatorial coverage as an aspect of test quality. *CrossTalk*, *28*(2), 19-23.
11. Chilenski, J. J. (2001). *An investigation of three forms of the modified condition decision coverage (MCDC) criterion. US Department of Transportation, Federal Aviation Administration*. DOT/FAA/AR-01/18 (April 2001).
12. Johnson, L. A. (1998). DO-178B, Software considerations in airborne systems and equipment certification. *Crosstalk, October*, *199*.
13. RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification," December 1, 1992.
14. Ferrell, T. K., & Ferrell, U. D. (2017). RTCA DO-178C/EUROCAE ED-12C. *Digital Avionics Handbook*.
15. D.R. Kuhn, I. Dominguez, R.N. Kacker and Y. Lei. "Combinatorial Coverage Measurement Concepts and Applications", *2nd Intl Workshop on Combinatorial Testing*, Luxembourg, IWCT2013, IEEE, Mar. 2013.
16. Kuhn, D. R., Kacker, R. N., & Lei, Y. (2016). Measuring and specifying combinatorial coverage of test input configurations. *Innovations in systems and software engineering*, *12*(4), 249-261.
17. Bartholomew, R. (2013, May). An industry proof-of-concept demonstration of automated combinatorial test. In *Proceedings of the 8th International Workshop on Automation of Software Test* (pp. 118-124). IEEE Press.

18. Leithner, M., Kleine, K., & Simos, D. E. (2018, April). CAmetrics: A tool for advanced combinatorial analysis and measurement of test sets. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 318-327). IEEE.