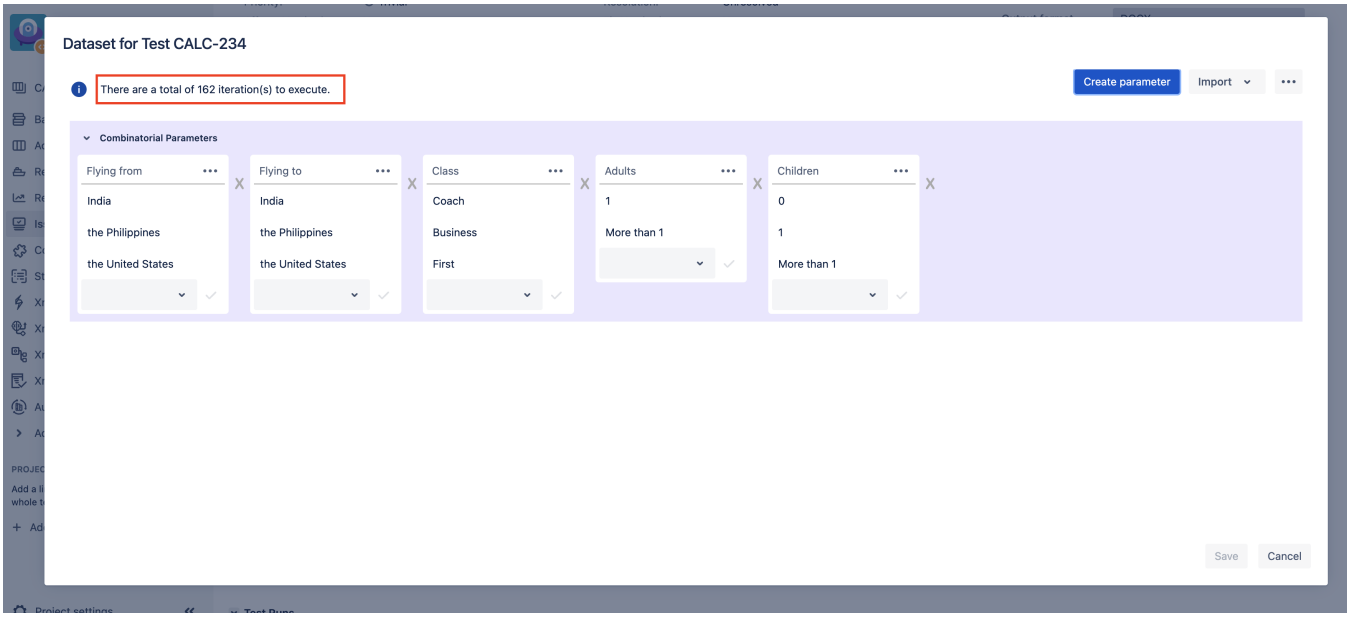
Advanced Testing Techniques To Improve Efficiency

Systems, in general, including software and hardware, usually have some input parameters that will affect the output(s) produced by the system. The number of parameters and the possible values, their ranges, vary and can be limited, huge but finite, or even infinite.

Taking a simple flight booking site as an example, we can easily have thousands of combinations for Flying From, Flying to, Class, (number of) Adults, and (number of) Children input parameters.



This potentially leads to a high number of scenarios to be tested - for the example model above, with a limited number of possible values, that would still lead to 3*3*3*2*3=162 scenarios.

In general, we could think that if we aim to test systems like this in-depth, we would need to test them with all the possible combinations of values of these parameters.

But does it even make sense? Are any of those scenarios redundant, or in other words, is there any manageable subset of scenarios that can be tested that can still help us find bugs?

In this tutorial, we'll learn about the testing challenges of these systems and how to overcome them efficiently.

> A system generates different results/outputs due to changes in the input parameters and uses the system in different contexts (e.g., configurations, operating system, time zones, cloud provider).
>
> Those types of variation ideas should also be accounted for in your testing.

Initial testing options

Option 1: Test using a “familiar” selection of values for the parameters

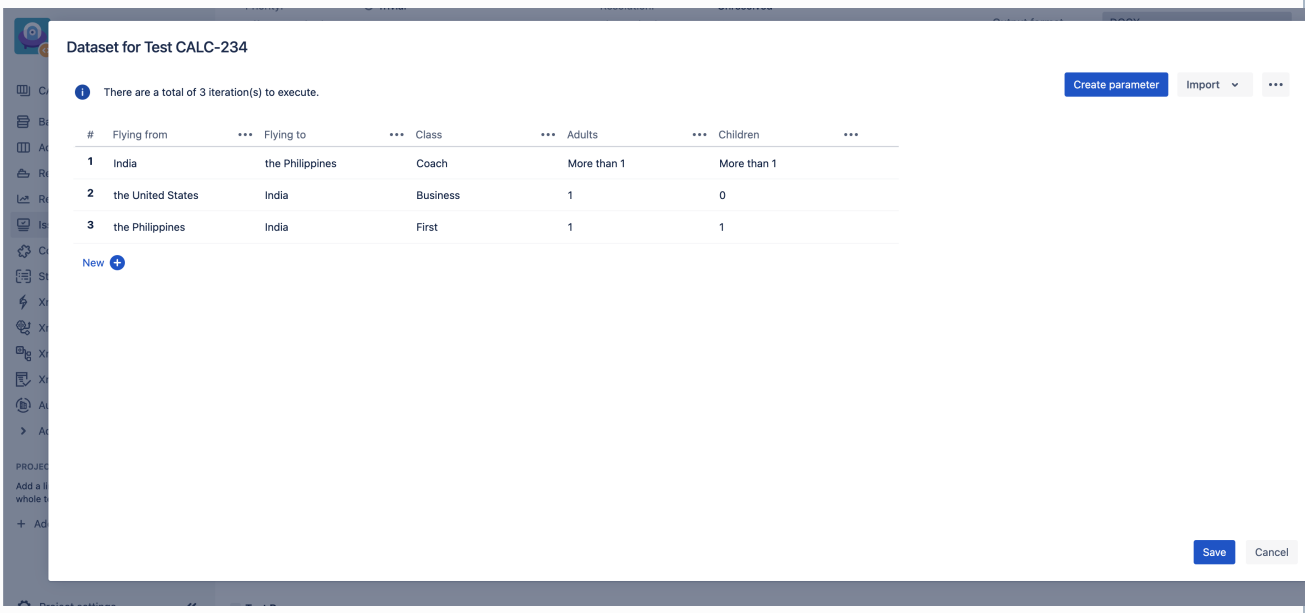
The first strategy that we may come up with is data-driven testing. It is a technique where a well-defined test script is executed multiple times, taking into account a "table" of parameters and corresponding values.

Usually, data-driven testing is used as a way to inject data to test automation scripts. Still, it can also be used to manually perform the same test multiple times against different data iterations.

However, the exact combination of parameter values to be used is beyond the scope of data-driven testing. Usually, testers include parameter value combinations that represent examples coming as a direct consequence of acceptance criteria, from well-known "happy paths," or from the production data.

Learn more

Xray has built-in support for datasets where testers can explicitly enumerate parameters and the combination of values to be tested.



Please see Parameterized Tests (Xray for Jira Server/DC), Parameterized Tests (Xray for Jira Cloud) for more info.

Option 2: Test using a random combination of parameter values

Random testing is always an option, but it doesn't ensure we test combinations that matter unless we perform a very high number of tests, which would probabilistically include a certain % of combinations or even all of them if we spend an infinite time randomly testing.

Nobody wants to perform testing endlessly without any criteria. Random testing doesn't ensure we cover combinations that matter with a very manageable set of tests.

Option 3: Test every parameter/value combination

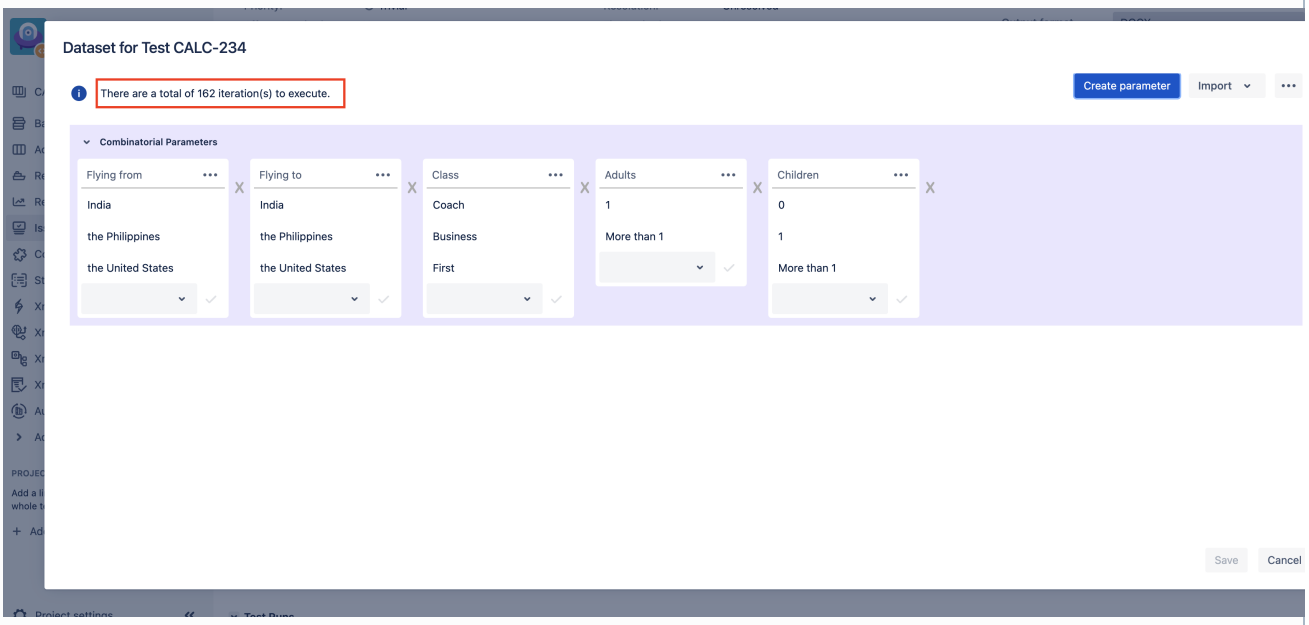
Testing every possible combination of parameters is only viable if we have very few parameters with very few possible values for each one of them.

In general, testing all combinations:

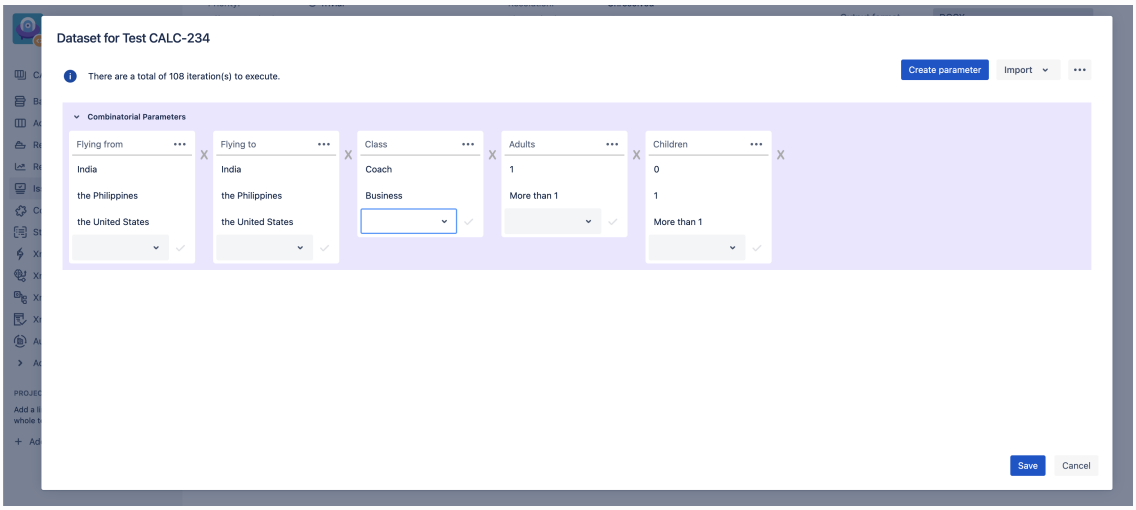
- takes considerable time
- is costly in terms of human resources or infrastructure

Learn more

Xray also supports combinatorial parameters, where the user defines the values for each parameter, and Xray calculates all the possible combinations, turning that into the dataset to be used.



It's possible to remove some values of the combinations to be generated. For example, we can exclude the "First" Class. That would lead to fewer scenarios to test (e.g., 162 => 108) but could still not be enough if we aim to have a limited set of tests.



Please see Parameterized Tests (Xray for Jira Server/DC), Parameterized Tests (Xray for Jira Cloud) for more info.

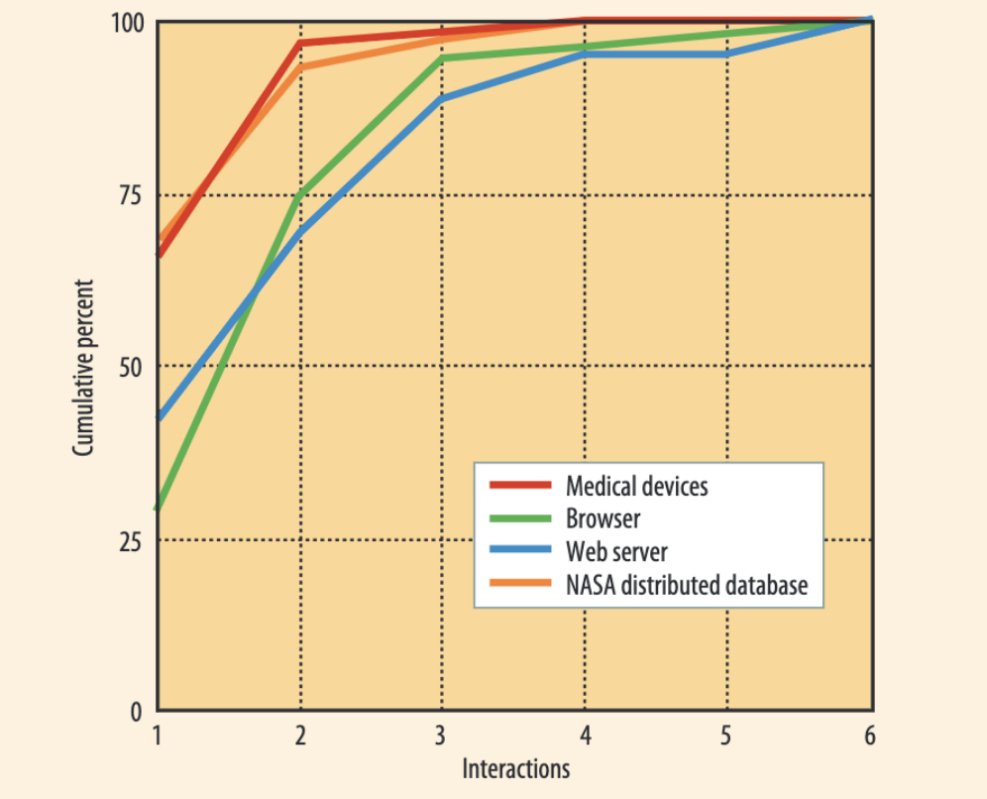
What empirical studies tell us about fault detection

Before we talk about the optimal testing option, let's look at the underlying practical evidence.

Studies such as "Estimating t-Way Fault Profile Evolution During Testing" and "Practical Combinatorial Testing" (presented by the National Institute of Standards and Technology in 2017 and 2010 respectively) indicate that the vast majority of defects (67%-93%) related to input values are due to a problem in either one parameter value (single-value fault) or a combination of two parameter values (2-way interaction fault).

NIST research showed that most software bugs and failures are caused by one or two parameters, with progressively fewer by three or more, which means that combinatorial testing can provide more efficient fault detection than conventional methods. Multiple studies have shown fault detection equal to exhaustive testing with a 20X to 700X reduction in test set size. New algorithms compressing combinations into a small number of tests have made this method practical for industrial use, providing better testing at lower cost. *- NIST*

The following chart (see reference ahead) represents the cumulative error detection rate for fault-triggering conditions.



The key insight underlying t-way combinatorial testing is that not every parameter contributes to every fault, and many faults are caused by interactions between a relatively small number of parameters *- Combinatorial Software Testing, Rick Kuhn and Raghu Kacker, National Institute of Standards and Technology, Yu Lei, University of Texas at Arlington, Justin Hunter, Hexawise*

Single-value faults are most likely due to typical mistakes, such as the *off-by-one* bug (e.g., imagine using a loop and using the "<" operator instead of "<="). The interaction of 2 parameters may be due to bugs around implementing cascade conditional logic statements (e.g., using *if* or similar*)* involving those variables.

Bugs related to the interaction of more parameters decrease with the number of parameters; in other words, finding these rare bugs will require much more tests to be performed, leading to more time/costs. However, those rare *t-way interaction faults* can also be critical and addressed by proper testing techniques.

Recommended Option: Combinatorial Testing considering 2-way (pairwise) and t-way interaction of parameters

Given the empirical data mentioned earlier, adopting combinatorial testing is an approach that provides great results in terms of fault detection/defect slippage prevention with a manageable test suite size.

It is a "black-box test technique in which test cases are designed to exercise specific combinations of values of several parameters" (ISTQB).

How do we come up with the exact combination of parameter values?

Instead of letting humans select them by hand, we rely on tools to perform them more efficiently. There are different algorithms for generating t-way interactions of parameters (e.g., pairs, triplets) - some may create more scenarios than others to achieve the same coverage (in terms of the interaction of parameters) and take more or less time. There are a couple of important algorithm features to consider, as seen ahead.

Reducing the number of test scenarios and the time to create them

The core function of combinatorial algorithms is identifying the smallest mathematically-possible set of scenarios to satisfy the t-way condition and doing that much faster than in the manual approach.

Imagining the previous example, instead of having 162 test scenarios to perform, we could have just 13 using the pairwise technique to generate them.

Sometimes, we may need to test more thoroughly and ensure that we cover, e.g., all combinations of 3 values across all parameters, which would still require only 35 scenarios.

We may even have risk-based use cases where combinations of certain parameters are tested more thoroughly than others.

Test scenarios are usually generated in a specific order so that the interaction coverage is greater with the first tests and lesser with the last ones. This way, if we stop testing at a given moment, we can ensure that we tested the most combinations possible; some tools, such as Test Case Designer, allow us to track exactly the coverage of combinations achieved with a given number of executed tests.

In sum, there is a balance between the number of tests we execute and the coverage of interactions between variables (i.e., "t-way coverage") we validate.

Combining business acumen with statistical techniques

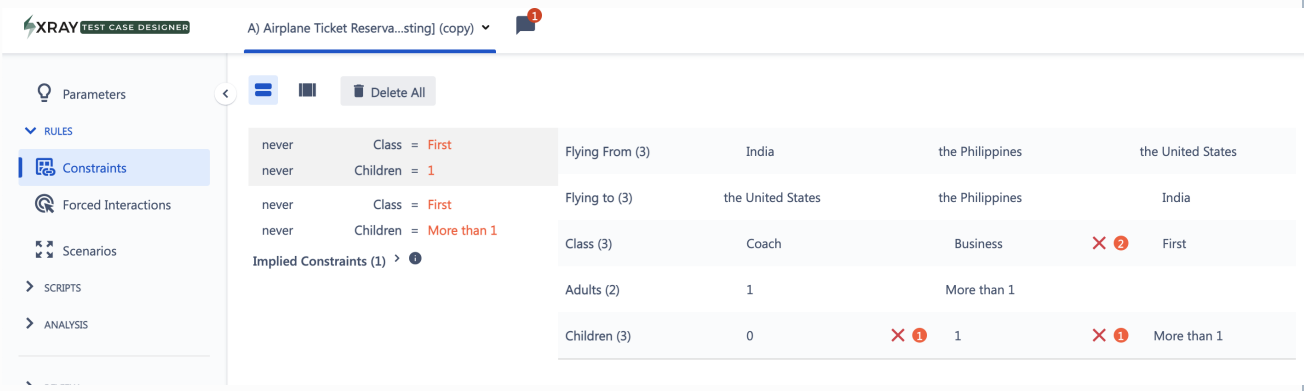
Even if we use pairwise testing, or n-wise testing in general, to dramatically reduce the number of test scenarios, not all of these combinations may make sense for several reasons. The statistical side of the algorithm would not automatically account for the subject matter expertise.

For example, the Departure and Destination values need to be different in our flight booking scenario. Also, we may have some rules where the First class is unavailable to children.

Therefore, the algorithm should support rule handling to limit specified interactions in the generated data set.

Example with Xray's Test Case Designer

In Test Case Designer, we can apply "constraints" involving the combination of 2 parameters. We can apply several constraints as shown in the following example: Class=First cannot exist together either with Children=1 nor Children=Mode than 1.



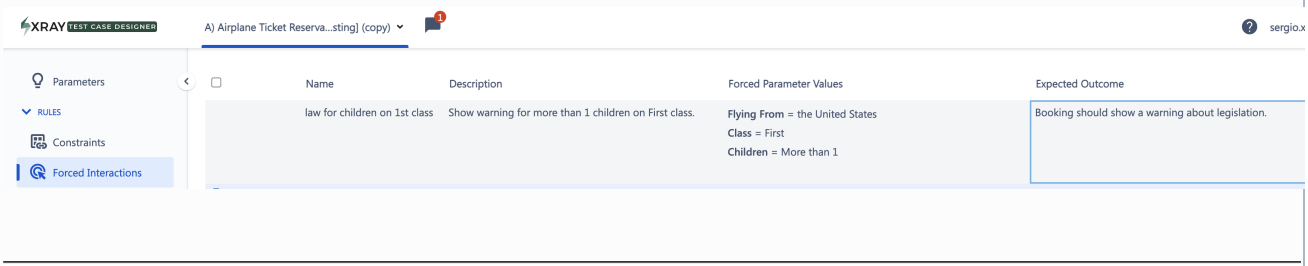
Further, not all combinations of parameters may be equally representative.

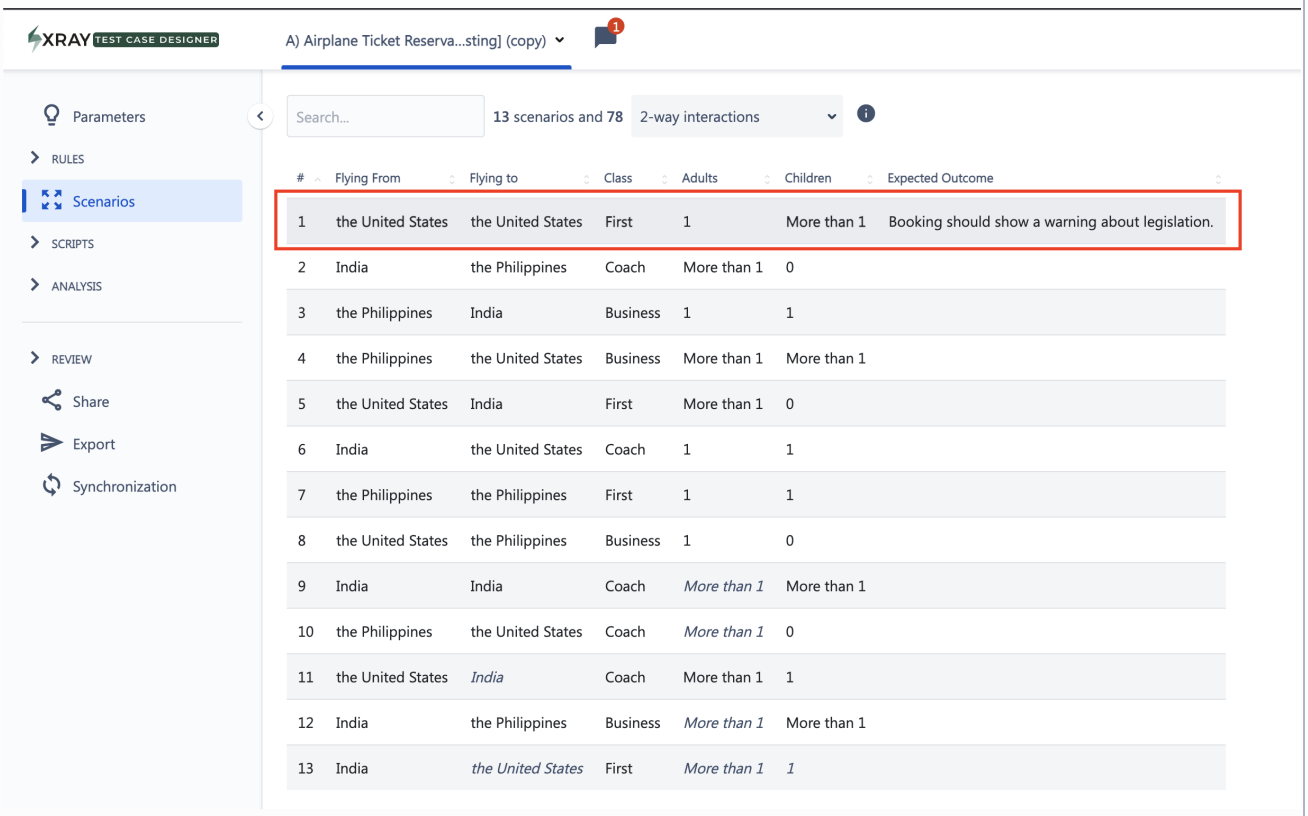
Sometimes there are highly important interactions as they represent frequently used happy paths or especially impactful previous defects.

The algorithm should allow users to intervene in the statistically-driven order and change the priority of certain scenarios.

Example with Xray's Test Case Designer

In Test Case Designer, we can achieve that by using Forced Interactions. In the following example, we consider a rule that we need to test due to a hypothetical legislation where some warning must be shown to users who are departing from the USA, using the First class, and have more than 1 child (i.e., 3 specific parameter values participating in 1 condition). That scenario will be added as the first row in the generated data set.





Using pairwise and t-way for scripted testing and exploratory testing

Whenever generating an optimized dataset (i.e., multiple "rows" of values for the parameters), this will be typically used to data-drive a scripted test case (e.g., a "manual" test composed of steps or an automated test script).

In that case, testers would specify the steps to follow and include references to the parameters of those steps. To perform testing, the test is iterated multiple times, as many as of the generated dataset rows (i.e., a combination of parameters/values). In each iteration, the parameters are replaced by the corresponding values on the dataset row.

Generating these combinations is useful not only for this testing approach, though.

Pairwise and t-way testing doesn't tell us how to perform testing; it just generates the combination of parameters. Therefore, we can also use this technique if we adopt a more exploratory testing approach, for example, for certain configurations of hardware/software.

Challenges

Pairwise, or t-way testing in general, even though useful, is not a silver bullet.

Some challenges or limitations to be aware of include the following:

1. **mindset shift:** thinking about a system from the perspective of parameters, values, and their interactions is significantly different from traditional testing techniques. Therefore, combinatorial testing has a fairly steep learning curve, but the investment pays off in the medium term with improvements in both efficiency and quality.
2. **model scope:** not all models are valuable, not all features have the same importance, and not all aspects of a feature have the same levels of risk. Even with the understanding of combinatorial methods, this testing approach requires significant collaboration between testers and business stakeholders to determine the right level of detail and priorities in each generated scenario (*such collaboration should happen regardless, but its importance is increased in combinatorial and model-based testing);*
3. **test oracle:** this technique doesn't address finding the proper test oracle for the generated scenarios. How do we know the scenario is behaving as expected? How do we know that a given scenario has issues or not?

Xray datasets and Xray Test Case Designer

Xray has built-in support for parameterized tests and datasets, supporting user-defined datasets and the automatic generation of combinations for the identified parameters.

Test Case Designer (TCD) provides a more comprehensive modeling tool where it's possible to:

- define parameters and values,
- enforce system-under-test rules,
- generate optimized datasets using 2-way or t-way settings, up to a certain level, within minutes.

With Test Case Designer, you can have a manageable set of test scenarios to perform and ensure that most combinations of values are covered early on in the test suite, so that most risk is addressed upfront.

TCD doesn't replace Xray's built-in capabilities for parameterized tests and datasets; it's a more evolved approach. Both can be used in a given project.

Test Case Designer is a feature only available to Xray Enterprise users.

	Xray's parameterized tests & datasets **(in all Xray versions)**	**Xray Test Case Designer** **(part of Xray Enterprise only)**
Parameters		
• define parameters	x	x
• parameters: enumerate possible values	x	x
• parameters: ranged values and equivalence classes	-	x
Dataset/scenarios generation		
• custom datasets (i.e., user-defined examples of parameter values)	x	-
• generation of all combinations of parameters/values	x	x (up to 6-way)

• generation of a partial combination of parameters	x	-
• generation of scenarios using pairwise (2-way testing)	-	x
• generation of scenarios using t-way testing (including risk-based settings)	-	x
• algorithmic enforcement of rules on the generation of scenarios	-	x
• forced interactions	-	x
Creation of tests using generated data		
• authoring test cases (definition of steps) using the generated data	x	x
• generation of test automation code skeleton for multiple testing frameworks using the generated data	-	x
Reporting		
• track t-way coverage	-	x