Abstract—Combinatorial Interaction Testing (CIT) has gained a lot of attention in the area of software engineering in the last few years. CIT problems have their roots in combinatorics. Mathematicians have been concerned with the NP-complete problem of finding minimal covering arrays (in other words, minimal CIT test suites) since early nineties. With the adoption of these techniques into the area of software testing, an important gap has been identified - namely consideration of real-world constraints. We show that indeed finding an efficient way of handling constraints during search is the key factor in wider applicability of CIT techniques.

I. INTRODUCTION

Many real-world software systems are highly configurable. It is usually infeasible to test all the possible configurations. In order to avoid running into this combinatorial explosion problem, Combinatorial Interaction Testing (CIT) techniques have been developed specifically for such systems. CIT combines all \( t \)-combinations of parameter inputs or configuration options in a systematic way so that we know we have tested a measured subset of the input or configuration space. Different parameter values can be set, for instance, via user interface. An example would be web browser configurations, as shown in Table I.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Load content & Notify pop-up & Cookies & Warn before & Remember downloads \\
\hline
Allow & Yes & Restrict & Yes & No \\
Restrict & No & Allow & No & \\
Block & & & & \\
\hline
\end{tabular}
\caption{WEB BROWSER CONFIGURATIONS}
\end{table}

Historically, CIT problems come from the field of combinatorics and are usually represented as a covering array (CA): \( CA(N; t, k_1, k_2, \ldots, k_m) \), where \( N \) is the size of the array, \( t \) is its strength, sum of \( k_1, \ldots, k_m \) is the number of parameters and each \( v_i \) stands for the number of values for each of the \( k_i \) parameters in turn. Suppose we want to generate a pairwise interaction test suite for an instance presented in Table I. It has 5 parameters, two of which can take 3 values (‘Allow’, ‘Restrict’ and ‘Block’), while others have two choices of values (‘Yes’ and ‘No’). The pairwise CIT problem is then formulated as: \( CA(N; 2, 3, 3) \). Furthermore, in order to test all combinations one would need \( 3 \times 2 \times 3 \times 2 \times 2 = 72 \) test cases. If, however, we cover all interactions between any two parameters, then we only need 9 test cases. Such a test suite is called a 2-way or pairwise test suite. The goal is to find the smallest covering array (or, in other words, CIT test suite) that covers all possible combinations of values, i.e. interactions between any set of \( t \) parameters.

Generation of a minimal CIT test suite is a very challenging task, in fact, the complexity is NP-complete. Therefore, exact methods, like the one by Hnich et al. from 2006 [1] have not been as successful as heuristics in real-world applications. There are several approaches for covering array generation.

II. CONSTRAINTS IN CIT

Even though CIT has been successful in discovering existing faults, in situations where there are hundreds of parameters like in the case of Software Product Lines [14], sometimes generation of even pairwise interaction test suite is a challenging task. Furthermore, such a test suite might be simply too big to be used in practice. Furthermore, until recently, higher-strength CIT testing, that is, 3-way, 4-way and higher, was deemed unfeasible in real-world situations [15].

However, in most CIT applications, the problem domain is constrained: some interactions are simply infeasible due to these constraints [1], [16], [17], [18]. The nature and description of such constraints is highly domain specific, yet taking account of them is essential in order for CIT to be usable in practice. Any CIT approach that fails to take account of constraints will produce many test cases that are either unachievable in practice or which yield expensively misleading results (such as false positives). An example of such a hard
**constraint** for the example shown in Table I is: “Warn before add-ons install” must always be turned on, that is, it can only take the “Yes” value.

Another type of constraint, often referred to as a soft constraint [16] may also have a role to play. Soft constraints are combinations of options that a tester believes do not need to be tested together (based either on their knowledge of the test subject and/or by a static analysis). Catering for such constraints will not improve test effectiveness, but it may improve efficiency. An example would be testing the “find” function that searches through a file looking for a particular pattern. If an empty file is supplied, it is not necessary to test searches of all possible patterns of words, since, given an empty file, an error should be thrown in each case. However, there has been little work on CIT with this type of soft constraint.

By prohibiting certain parameter-value interactions, constraints may significantly reduce the search space for a CIT test suite generation algorithm. This has been exemplified in recent work [15]. In particular, it has been shown that it is the combination of soft and hard constraints that allows for a very popular CIT-test generation algorithm, based on simulated-anneling, to scale to real-world problems. Furthermore, it made higher-strength testing, in particular, testing of all value-combinations of any 5 parameters, feasible.

**III. FUTURE RESEARCH DIRECTIONS**

Even though Combinatorial Interaction Testing is a relatively mature research area, the constraint handling techniques leave room for improvement. Last year’s survey of CIT tools [5] reveals that many tools simply do not implement any constraint handling method. One of the challenges has been efficient integration of constraint handling within the search process. For example, when using a heuristic such as simulated-anneling, a test case mutation can result in constraint violation. Therefore, an additional check needs to be made.

Arguably the first method deployed is the one that filters forbidden interactions. In other words, a list of disallowed parameter-value combinations is kept in memory. Each generated test case is then verified against this set of forbidden assignments and/or disallowed interactions are prevented during the search process. For example, the greedy-based ACTS tool [19], implements this method.

Alternatively one can choose a Boolean satisfiability or constraint solver to resolve constraints [2], [20]. However, these tools have been used as black-boxes. Especially in the case of constraint solvers, the right configuration might drastically speed-up the solving process.

We believe that further research into constraint handling methods during search in generation of Combinatorial Interaction Testing test suites can not only speed-up existing tools, but also reveal new algorithms efficient for higher-strength CIT testing.

**REFERENCES**


