Towards Model-Driven Test Case Concretization for End-to-end Combinatorial Testing

Léna Bamouh, Erwan Bousse

Towards Model-Driven Test Case Concretization for End-to-end Combinatorial Testing

Léna BAMOUH
lena.bamouh@etu.univ-nantes.fr
Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004
Nantes, France

Erwan BOUSSE
erwan.bousse@univ-nantes.fr
Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004
Nantes, France

Abstract

Combinatorial testing can be used to automatically generate relevant sets of combinations of abstract test data for a System Under Test (SUT). It requires defining a combinatorial model with possible abstract values for the SUT input parameters, from which relevant abstract test cases can be generated to reach a chosen coverage criterion. However, concretizing abstract test cases into concrete test cases, and writing corresponding test scripts, is a tedious and error-prone manual process. With a focus on Java unit testing, we present in this paper a first end-to-end approach where combinatorial testing is supplemented with a model-driven *concretization* step for abstract test cases. To produce concrete test cases out of abstract test cases, the process requires a context-specific data generator provided by the test engineer, which can be implemented using constraint solving techniques. A code generator is used to produce the working JUnit test script for each concrete test case. The approach is implemented and integrated with the PICT combinatorial testing tool, the Choco-Solver Java library for data generation, and the Eclipse Modeling Framework (EMF) for model management. While the approach is currently limited to primitive data types, an initial evaluation on five Java methods shows that the testing effort can be greatly reduced when the combinatorial complexity is high.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; *Model-driven software engineering*; • **General and reference** → **Verification**; • **Theory of computation** → Constraint and logic programming.

Keywords

Combinatorial testing, Unit testing, Automatic test generation, Test case concretization

ACM Reference Format:
Léna BAMOUH and Erwan BOUSSE. 2024. Towards Model-Driven Test Case Concretization for End-to-end Combinatorial Testing. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3652620.3687823

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3687823

1 Introduction

In the field of software testing, combinatorial testing [8] stands as a systematic approach to efficiently explore a subset of possible input combinations of a given System Under Test (SUT). First, through a careful analysis of the specification of the SUT, a combinatorial model is defined as an abstraction of the different input parameters of the SUT. This model is composed of a set of abstract parameters and values that reflect interesting cases of input parameters (e.g., given an input parameter $l \in List$, an abstract parameter $x \in \{$*"l is empty"*, *"l is not empty"*$\}$ puts the emphasis on the absence or presence of values in l). Then, using an algorithm such as IPOG[9], a set of abstract test cases can be automatically generated from a combinatorial model using a target coverage criterion (e.g., covering each pair of possible abstract values, commonly known as *pairwise* coverage). This results in an abstract test suite that efficiently covers a subset of possibilities defined in the combinatorial model, and can thus satisfyingly cover the SUT specification.

However, while abstract test cases can be automatically generated, two remaining ensuing steps are commonly performed manually. First, each abstract test case must be translated into a *concrete test case*, where a concrete value is assigned to each SUT input parameter. This step is not trivial, as not only must the chosen inputs satisfy constraints of the abstract test case (e.g., given an abstract value $x =$ *"l is not empty"*, a possible concrete value for l is $[1, 3, 5]$), but an oracle consistent with the SUT specification must be manually defined and added to the concrete test case. Second, a test script must be implemented with a testing framework (e.g., JUnit[1]) for each concrete test case. These two steps represent costly, tedious and error-prone manual work, especially given an important amount of test cases.

To cope with this problem, we present in this paper a first end-to-end approach where combinatorial testing is supplemented with a model-driven *concretization* step for abstract test cases. The proposed solution focuses on Java unit testing, i.e., the SUT is assumed to be a Java method. First, an abstract test suite model is generated from the output of an existing combinatorial testing tool named PICT[2]. Then, the abstract test suite model is transformed into a concrete test suite model using a data generator provided by the test engineer. This data generator is specific to the considered tested Java method, and can be implemented using constraint solving tools. A code generator is finally used to transform the concrete

[1] https://junit.org/
[2] https://github.com/microsoft/pict

test suite model into a set of working Java JUnit test scripts. In addition, we propose a way to specify simple oracles directly within the combinatorial model, thus allowing such oracles to be automatically generated in the abstract test suite. These oracles are then eventually propagated in the concrete test suite and test scripts.

We implemented a prototype using the Eclipse Modeling Framework (EMF) for model management and JavaPoet for code generation. We performed an initial evaluation with five Java methods, for which data generators were developed using the Choco-Solver library [11]. Results show that when the combinatorial complexity is important, the reduction of the concretization effort outweighs the overhead of writing a data generator.

The remainder of the paper is structured as follows. Section 2 provides the required background. Section 3 details our contribution, an i.e., a model-driven approach to concretize abstract test cases. The approach is evaluated using our prototype in Section 4. Section 5 lists previous works related to our research topic. Finally, Section 6 concludes the paper by summarizing its contributions and suggesting avenues for future research.

2 Background

In this section, we present an overview of combinatorial testing and constraint programming, along with a running example.

2.1 Combinatorial Testing

2.1.1 Principles. Combinatorial testing [8] aims at efficiently identifying relevant combinations of input arguments for a given System Under Test (SUT), which can then be used to define test cases for the SUT. The main idea is to create a simpler but well-founded abstraction of the input domain of the SUT, and to derive from this abstraction a reasonable amount of test cases that cover the SUT specification satisfactorily. A combinatorial testing process usually unfolds as follows:

(1) A careful analysis of the specification of the SUT is performed, in order to identify in which ways the SUT *input parameters* may influence the outcome of the execution.
(2) The result of this analysis is reified as an explicit *combinatorial model* composed of a set of *abstract parameters*. An abstract parameter describes an interesting characteristic of the SUT input parameters, and is associated with a domain of possible *abstract values*. Abstract parameters are expected to have much smaller domains of values than input parameters, thus reducing the complexity of the testing problem. Note that abstract parameters can arbitrarily differ from input parameters, both in numbers and in nature, as the running example will show.
(3) Optionally, a set of constraints is defined, to allow/forbid specific abstract values under specific conditions.
(4) A target *coverage criterion* is chosen, which specifies how abstract values should be covered by the test suite to be generated. The idea is both to limit the amount of test cases to generate, and to ensure a certain level of quality for generated test cases. A common family of criteria are called *n-wise coverage*, which are satisfied if the test suite covers each possible group of n abstract values of the combinatorial model. With $n = 2$, the criterion is commonly known as *pairwise coverage*.

```java
/**
 * Check whether a word is a palindrome.
 *
 * @param word The word to check.
 * @return true if word is a palindrome, false otherwise.
 * @throws IllegalArgumentException if word is either null, empty,
 *                                  contains a special character,
 *                                  or contains a digit
**/
static boolean isPalindrome(String word)
                    throws IllegalArgumentException
```

Listing 1: Specification and signature of `isPalindrome`

(5) An algorithm such as IPOG[9] is used to automatically generate an *abstract test suite* composed of a set of *abstract test cases* (ATC). Each abstract test case is composed of a set of exactly one abstract value per abstract parameter of the combinatorial model.
(6) Each abstract test case is manually translated into a *concrete test case* (CTC) composed of two parts: (a) test data, with a value assigned to each input parameter; (b) an oracle that specifies whether a resulting execution is a success.

To better illustrate this process, an example is given below in Section 2.1.3. Also, note that while defining a combinatorial model can be an intricate endeavor (especially given the SUT specification), we assume in this paper that the combinatorial model is always valid and well-defined, and we consider the prior task of defining this model outside the scope of the proposed approach.

2.1.2 PICT. Pairwise Independent Combinatorial Testing (PICT), is an open-source command-line tool developed by Microsoft. It provides a small textual language to define combinatorial models, and a command-line tool to generate abstract test cases from a given PICT combinatorial model. Any n-wise coverage criteria can be used, with pairwise being the default. Abstract test cases can either be printed as raw text, or can be exported as JSON for better integration with other processes. An example of a PICT combinatorial model is given below in Section 2.1.3.

2.1.3 Combinatorial Testing Applied to Unit Testing. While combinatorial testing can be applied to any sort of well-specified SUT, one possibility is to use it for *unit testing*, where the SUT takes the form of a software function or method. While the process remains identical, an additional step is then required : for each concrete test case, an executable *test script* must be manually implemented, i.e., a small program that calls the tested method with the test data specified in the test case, and that checks whether the execution is a success as defined in the oracle.

Example. Listing 1 shows the signature and specification of a Java method named `isPalindrome`, which verifies whether a given word is a palindrome. The specification states that (1) `true` should be returned if the input string is a palindrome, and `false` otherwise, and (2) an `IllegalArgumentException` should be thrown if the word is `null`, empty, contains digits or special characters.

Listing 2 shows a PICT combinatorial model to test `isPalindrome`. Given that the outcome of the method is entirely based on the contents of the input string, five abstract parameters are defined, each

```
# Abstract parameters and values
WordIsNull: yes, no
WordIsEmpty: yes, no, _
WordHasSpecialCharacter: yes, no, _
WordHasDigit: yes, no, _
WordIsPalindrome: yes, no, _

# Constraints
IF [WordIsNull] = "yes"
THEN [WordIsEmpty] = "_"
ELSE [WordIsEmpty] <> "_";

IF [WordIsEmpty] <> "no"
THEN [WordHasSpecialCharacter] = "_"
    AND [WordHasDigit] = "_"
    AND [WordIsPalindrome] = "_"
ELSE [WordHasSpecialCharacter] <> "_"
    AND [WordHasDigit] <> "_"
    AND [WordIsPalindrome] <> "_";
```

Listing 2: PICT combinatorial model for `isPalindrome`, manually derived from the specification shown in Listing 1

	WordIsNull	WordIsEmpty	WordHasSpecialCharacter	WordHasDigit	WordIsPalindrome
ATC1	*yes*	_	_	_	_
ATC2	*no*	*yes*	_	_	_
ATC3	*no*	*no*	*no*	*no*	*no*
ATC4	*no*	*no*	*no*	*yes*	*yes*
ATC5	*no*	*no*	*yes*	*no*	*yes*
ATC6	*no*	*no*	*yes*	*yes*	*no*

Table 1: Abstract test suite generated with PICT using the combinatorial model from Listing 2, shown in tabular form.

describing a specific important characteristic of said string. As each abstract parameter is defined as a closed question, their domains always include abstract values labeled `yes` and `no`. In addition, an abstract value with the underscore symbol _ is part of the domain of all parameters but `WordIsNull`. In this model, this symbol denotes the case where no meaningful value can be assigned to the parameter. Lastly, a set of constraints are defined in order to force or forbid specific abstract values in specific circumstances. The first constraint states that if `WordIsNull` equals `yes`, then the only possible value for `WordIsEmpty` is _, as the "emptiness" of the string is not meaningful in this case. Similarly, the second constraint states that if `WordIsEmpty` is not false (i.e., it is either `yes` or _), then all other parameters take the value _, as constituents of the string are not relevant if it is empty or `null`.

Table 1 shows an abstract test suite obtained after running PICT on the combinatorial model using the pairwise coverage criterion as a target. Six abstract test cases are yielded: ATC1 corresponds to a `null` input; ATC2 to an empty input; ATC3 to an input that is not `null`, not empty, without a special character, without a digit, and that is not a palindrome; and so on. From there, each ATC has yet to be translated first into a concrete test case (i.e., with test data and an oracle), then into an executable test script.

2.2 Constraint Programming

Constraint Programming (CP)[4] is an alternative to traditional programming, focusing on defining constraints over variables and domains rather than using step-by-step algorithms. CP uses constraints to specify acceptable combinations of values from variable domains, framing problems through a series of constraints and then identifying values that satisfy all constraints. This latter task is performed by a constraint solver by efficiently navigating the domains to identify feasible assignments.

Different libraries and tools can be used for CP, such as Choco-Solver[11], a robust Java-based framework for modeling and solving constraint satisfaction problems. It allows the definition of problems using decision variables, domains, and constraints through its API.

A possible use of CP is for data generation tasks requiring specific formats or values[6]. In the approach presented thereafter, we demonstrate how CP can be used to generate the test data necessary to translate abstract test cases into concrete test cases.

3 Model-driven Test Case Concretization

In this section, we present a model-driven approach to concretize abstract test cases obtained with combinatorial testing. We first give an overview of the proposed solution, then we present in detail the process to import abstract test cases, before explaining how they are concretized and transformed into executable test scripts. Finally, we describe the implemented prototype.

3.1 Overview

Figure 1 shows an overview of our proposed approach. In the bottom right corner, we assume the software developer prepared a Java method that should be tested. In this first version of this work, we assume the Java method to be static and to only rely on primitive types (integers, strings, booleans, enumerations) for its input parameters and return value.

On the left, we assume the software tester decides to rely on combinatorial testing for this task, and to rely on the proposed approach to automatically obtain a JUnit test script at the end of the process. The *Abstract Test Case Generation* stage starts with a PICT combinatorial model realized by the software tester, and extended to include a definition of simple test oracles when possible. Then, PICT is executed (1) with a given coverage criterion in order to produce a set of PICT abstract test cases in JSON format. In parallel, the PICT combinatorial model is imported (2) in the form of a tool-independent combinatorial model conforming to an Ecore combinatorial metamodel. Finally, the PICT abstract test cases are also imported (3) in the form of an abstract test case (ATC) model conforming to an Ecore ATC metamodel. This ATC model includes cross-references to the combinatorial model imported in (2).

On the right, the *Test Case Concretization and Test Code Generation* stage includes the two core steps of the proposed approach. First, the concretization step (4) aims at transforming the ATC model into a concrete test case (CTC) model conforming to an Ecore ATC metamodel. This step requires the software tester to implement

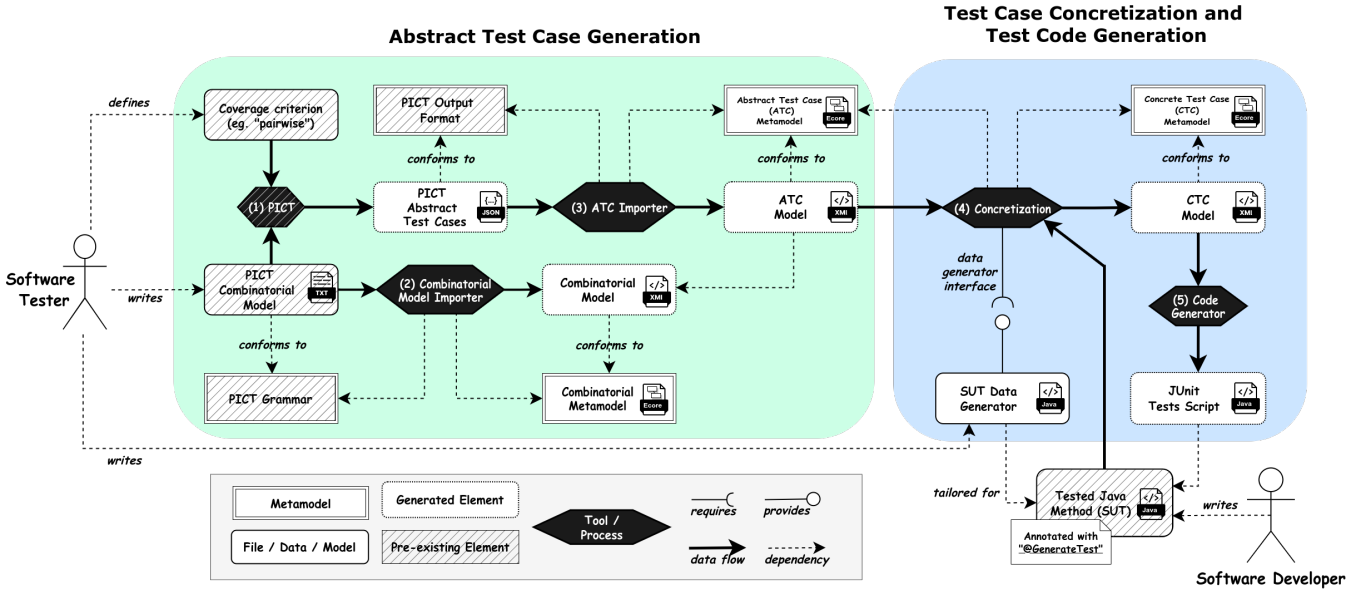

Figure 1: Overview of the proposed approach

and provide a data generator specific to the tested Java method. This data generator, typically implemented using constraint programming, is used to automatically produce valid test data for each source ATC. Finally, a code generator (5) is used to automatically produce a JUnit test script containing all test cases defined in the CTC model. If some oracles could not be handled by the automated process, some manual work may be required to finish the test script.

Figure 2 shows all the three metamodels defined for the proposed approach: the combinatorial metamodel (`CombinatorialMM`), the abstract test case metamodel (`ATCMM`), and the concrete test case metamodel (`CTCMM`). Elements of these metamodels are presented in later sections when they are required.

3.2 Abstract Test Case Generation

The first stage of the proposed process is dedicated to defining an extended PICT combinatorial model, followed by generating and importing abstract test cases into a proper model.

3.2.1 Extended PICT combinatorial model definition. As already presented in Section 2.1.2, we assume the combinatorial model to be defined using PICT. Such a model is composed of a set of abstract parameters, each capturing an important facet of the SUT input domain. From there, each generated abstract test case will assign an abstract value to each abstract parameter.

As abstract test cases do not work with concrete test data, a consequence is that they do not include the definition of an oracle that would specify conditions the concrete output data should fulfill. This explains why the translation of an abstract test case into a concrete test case requires some additional work to manually define the yet missing oracle. To address this limitation, our approach offers the possibility to define simple oracles directly within the combinatorial model, when relevant for the considered SUT. This enables the generation of abstract test cases with oracles, which can then be used to generate complete test cases in later stages.

The required extra abstract parameter in the PICT combinatorial model must be named `Oracle`. Possible values for this parameter must follow the following textual syntax:

- `return(`*`Java expression`*`)` specifies an oracle that checks that the returned value is equal to an arbitrary Java expression.
- `throws(`*`Java exception type`*`)` specifies an oracle that checks that an exception of a specific Java type was thrown.
- `undefined` specifies that the oracle cannot be defined in this case. This should for instance be used when the expected return value is variable and not constant.

In addition, to specify in which cases a specific oracle should be used, a set of strict constraints must be added to the model. Altogether, these constraints should enforce that only a single possible oracle value may be generated for a given set of abstract values.

Listing 3 shows an example of oracle definition for the PICT combinatorial model prepared for `isPalindrome` previously shown in Listing 2. An `Oracle` abstract test parameter is defined with three possible values assessing that the returned value should be `true` or `false`, or that an `IllegalArgumentException` should be thrown. Next, three constraints are defined enforcing in which cases each of these values should be used.

Next, PICT can be executed on the extended combinatorial model with a target coverage criterion to produce an abstract test suite, which we eventually import as a proper model, as explained below.

3.2.2 ATC and Combinatorial Model Import. The left of Figure 2 shows the two metamodels used in this stage. In the top-left corner, `CombinatorialMM` is used to represent a combinatorial model as a set of `AbstractParameters`, each with a domain of `AbstractValues`. Constraints are not considered as they are not required in later stages. This metamodel is used as a target format when importing

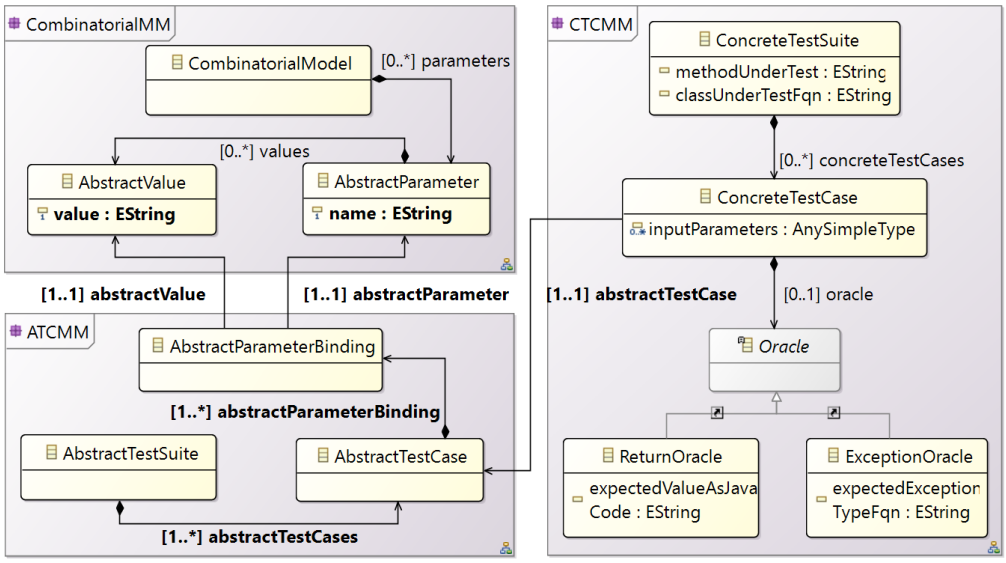

Figure 2: Metamodels used in the approach: at the top-right, the combinatorial metamodel; at the bottom-left, the abstract test case metamodel; on the right, the concrete test case metamodel

```
Oracle: return(true), return(false),
     throws(IllegalArgumentException)

IF [WordIsNull] = "yes"
    OR [WordHasSpecialCharacter] = "yes"
    OR [WordHasDigit] = "yes"
    OR [WordIsEmpty] = "yes"
THEN [Oracle] = "throws(IllegalArgumentException)"
ELSE [Oracle] <> "throws(IllegalArgumentException)";

IF [WordIsPalindrome] = "yes"
    AND [WordHasSpecialCharacter] = "no"
    AND [WordHasDigit] = "no"
THEN [Oracle] = "return(true)"
ELSE [Oracle] <> "return(true)";

IF [WordIsPalindrome] = "no"
    AND [WordHasSpecialCharacter] = "no"
    AND [WordHasDigit] = "no"
THEN [Oracle] = "return(false)"
ELSE [Oracle] <> "return(false)";
```

Listing 3: Definition of the `Oracle` abstract parameter and associated constraints for `isPalindrome`, extending the combinatorial model from Listing 2

the PICT textual combinatorial model into a proper model. Due to the very simple textual syntax for declaring abstract parameters in PICT, this can be accomplished using simple regular expressions.

In the bottom-left corner, `ATCMM` is the metamodel used to represent abstract test cases. An `AbstractTestCase` is composed of `AbstractParameterBindings`, each assigning an `AbstractValue` to an `AbstractParameter`. This assignment relies on cross-references with the `CombinatorialMM`, i.e., each `ATCMM` model instance requires a corresponding `CombinatorialMM` instance. This second metamodel is used as a target format when importing the abstract test cases, that PICT generates in JSON format, into a proper model.

For example, when importing abstract test cases generated by PICT using the `isPalindrome` combinatorial model shown in Listing 2, an `AbstractTestSuite` model is obtained. This model is composed of a set of `AbstractTestCase` elements, each containing the same information as one line of Table 1 (i.e., each column contains all `AbstractParameterBindings` of a specific `AbstractParameter`). If the combinatorial model is extended with the contents shown in Listing 3, abstract values are also generated for the `Oracle` parameter, as shown in Table 2 (which extends Table 1).

	`Oracle`
ATC1	`throw(IllegalArgumentException)`
ATC2	`throw(IllegalArgumentException)`
ATC3	`return(false)`
ATC4	`throw(IllegalArgumentException)`
ATC5	`throw(IllegalArgumentException)`
ATC6	`throw(IllegalArgumentException)`

Table 2: Extension of Table 1, adding one extra `Oracle` abstract parameter due to the model extension shown in Listing 3.

3.3 Concretization of Abstract Test Cases

The second stage of the approach focuses on the concretization of the ATC model through test data generation, CTC model generation, and code generation.

```java
public class PalindromeDataGenerator {
    public static String generateData(String nullWord,
            String emptyWord, String specialCharWord,
            String digitCharWord, String generatePalindrome) { ... }
}
```

Listing 4: Data generator's signature for `isPalindrome`. Each parameter of the generator corresponds to an abstract parameter of the combinatorial model shown in Listing 2.

3.3.1 Data generator interface. By definition, an abstract test case only defines the characteristics that input values should present, but does not define concrete input values that would allow the SUT to be executed. In the proposed approach, we propose to accomplish this concretization step in a systematic fashion by automatically generating test data that satisfies the constraints of a given ATC. This requires the software tester to provide a *data generator* specific to the tested method. The data generator must comply with an interface (i.e., a set of contracts) specified as follows:

- It must be defined in a Java public class as a public and static Java method.
- There must be one parameter of type `String` per abstract parameter of the combinatorial model, defined in the same order.
- The return type must match the input parameters of the tested Java method. If the tested method has a single input parameter, then the return type of the data generator must have the same type. If the tested method has multiple input parameters, then the return type must be of type `List<Object>`.

Listing 4 shows an example of data generator class and static method declarations for `isPalindrome`. Each parameter corresponds to one abstract parameter shown in the combinatorial model from Listing 1. Since `isPalindrome` only has one string input parameter, the return type is of type `String`.

Then, for each ATC, the data generator will be called using the ATC abstract values, in order to produce input parameters. The concretization step is explained in more details in Section 3.3.3.

3.3.2 CP-based Data Generator Implementation. Although the implementation of the data generator is rather open—as long as the above requirements are fulfilled—our initial work focused on the use of constraint programming (CP) to generate test data. More specifically, we investigated the use of the Choco-Solver Java library, which provides facilities to explicit constraints over variables and domains, and to generate values satisfying said constraints.

We consider that a CP-based data generator can be defined using the following design guidelines:

- Specify one CP *variable* per SUT input parameter. CP constraints will be applied on the variable, then a solver will be used to identify candidate values for the variable that satisfy said constraint.
- Define a CP *constraint* for each abstract value of a combinatorial model that implies a restriction on one or multiple SUT input parameter—e.g., if the abstract value is "the list l is not empty", then the constraint must be applicable to l and must ask the CP solver that $l.size$ should be above zero.
- When the Java method that generates data is called, it should go over each abstract value and apply the corresponding CP constraint on the CP variable of each involved input parameter. Then, the CP solver should find a solution for each CP variable, and return the produced test data.

```java
private static void addPalindromeConstraint(Model model,
        IntVar[] word) {
    for (int i = 0; i < (word.length + 1) / 2; i++)
        model.arithm(word[i], "=", word[word.length - 1 - i])
          .post();
}
```

Listing 5: CP constraint defined in the `isPalindrome` data generator ensuring that the generated word is a palindrome

```java
private static void addASCIIConstraints(Model model,
        List<IntVar> constraints, IntVar[] word,
        int lower, int upper, int start, int end) {
    for (int i = start; i < end; i++) {
        if (!constraints.contains(word[i])) {
            model.arithm(word[i], ">=", lower).post();
            model.arithm(word[i], "<=", upper).post();
            constraints.add(word[i]);
        }
    }
}
```

Listing 6: CP helper method in the `isPalindrome` data generator ensuring that certain letters of a string correspond to a specific range of ASCII codes

```java
private static void addDigitsConstraint(Model model,
        List<IntVar> constraints, IntVar[] word) {
    addASCIIConstraints(model, constraints, word,
                    48, 57, word.length/4, word.length/2);
}
```

Listing 7: CP constraint defined in the `isPalindrome` data generator to generate a word containing digits

For example, applying these guidelines to define a data generator for `isPalindrome` requires defining one CP variable for the `word` input parameter, and six CP constraints (one per `yes` abstract value and one for `no` abstract value for the `WordIsPalindrome` abstract parameter). Because Choco-Solver is not able to directly manipulate string variables, the CP variable for `word` is actually defined as an array of CP *integer* variables (`IntVar[]`), where each variable corresponds to a letter represented with its corresponding ASCII code[3]. For this example, we arbitrarily fixed the size of the array to five letters (i.e., five CP integer variables). Once integer values have been generated by the CP solver, each integer can be transformed into a character, and thus the array can become an actual string.

Listing 5 shows a CP constraint corresponding to the `yes` abstract value of the `WordIsPalindrome` abstract parameter. It specifies a set of CP constraints stating that each CP integer variable must be equal to its symmetrical counterpart.

Listing 7 shows a CP constraint corresponding to the `yes` abstract value of the `WordHasDigit` abstract parameter. It relies on a

[3] https://en.wikipedia.org/wiki/ASCII

Algorithm 1: Transformation of an `AbstractTestSuite` into a `ConcreteTestSuite`

```
Inputs:
abstractTestSuite: instance of AbstractTestSuite, from ATCMM
dataGenerator: data generator for the Java method under test
begin
    CTS ← createEmptyConcreteTestSuite()
    foreach abstractTestCase ∈ abstractTestSuite.abstractTestCases do
        CTC ← createEmptyConcreteTestCase()
        CTC.abstractTestCase ← abstractTestCase
        generatedData ← getDataForTestCase(dataGenerator,
          abstractTestCase)
        CTC.callArguments.addAll(generatedData)
        CTC.oracle ← createOracleFrom(abstractTestCase)
        CTS.concreteTestCases.add(CTC)
    end
    return CTS
end
```

	Input	Oracle	
	`word : string`	Return	Exception
CTC1	`null`	_	`IllegalArgumentException`
CTC2	`""`	_	`IllegalArgumentException`
CTC3	`"baaaa"`	`false`	_
CTC4	`"a0a0a"`	_	`IllegalArgumentException`
CTC5	`"!aaa!"`	_	`IllegalArgumentException`
CTC6	`"!0a0#"`	_	`IllegalArgumentException`

Table 3: `ConcreteTestSuite` model generated from the `AbstractTestSuite` corresponding to Tables 1 and 2. Each row represents a `ConcreteTestCase`, and each column gives either the input parameter or the oracle.

helper method shown in Listing 6 that is able to restrict a specific range of CP variables to a specific range of ASCII codes. Given that characters corresponding to integers (i.e., `0`,`1`,`2`,...) are found between codes 48 and 57 of the ASCII standard, the presence of a digit in the word is achieved by constraining between CP variables these two bounds.

3.3.3 Concrete Test Suite Model Construction. Using a data generator, the next step is to translate a given abstract test suite model into a *concrete* test suite model. The right part of Figure 2 depicts the `CTCMM` metamodel used for this purpose. It specifies a `ConcreteTestSuite` as a set of `ConcreteTestCases`, each containing a set of `inputParameters` and an `Oracle`. Aligned with the combinatorial model extension proposed previously, two types of oracles are supported: a `ReturnOracle` corresponds to checking that the method returns a specific value, and an `ExceptionOracle` specifies that a specific exception must be thrown by the method.

Algorithm 1 describes the transformation from an `AbstractTestSuite` model to a `ConcreteTestSuite` model. In a nutshell, the process calls the provided data generator for each provided ATC, and produces a corresponding `ConcreteTestCase` that includes the generated test data. An operation labeled *createOracleFrom* denotes the translation of an `Oracle` abstract value into a proper `ReturnOracle` or `ExceptionOracle` depending on the value textual content. If the abstract value of the `Oracle` parameter is `undefined`, *createOracleFrom* simply produces no value.

```
1 private static CodeBlock generateEqualCode(ReturnOracle oracle,
2         Method testMethod, Object callArguments) {
3     return CodeBlock.builder()
4       .addStatement("actualValue = $L($L)", testMethod.getName(),
5                     printGeneratedTestData(callArguments))
6       .addStatement("assertEquals($L, actualValue)",
7                     oracle.getExpectedValueAsJavaCode())
8       .build();
9 }
```

Listing 8: Excerpt of the test script code generator, where a test method's body for a concrete test case with a return oracle is generated

Table 3 shows an example of `ConcreteTestSuite` obtained for `isPalindrome` using the `AbstractTestSuite` corresponding to Tables 1 and 2 as input, and using the CP-based data generator partially presented above. Through the application of constraints based on the abstract values of the ATCs, the solver automatically discovers satisfying values for the `word` parameter. For instance, CTC6 corresponds to ATC6, which specifies that the input word should contain both a special character and a digit, while not being a palindrome, thus yielding the value `"!0a0#"`. Oracles are directly instantiated based on the contents of Table 2, e.g., for CTC6 this means expecting a `IllegalArgumentException` due to having an invalid input.

3.4 Test Scripts Code Generation

Once we have a concrete test suite model, the next and last step is to transform it into executable test scripts. We consider JUnit as a target testing framework for the generated scripts. In essence, each concrete test case must be turned into a JUnit method where the tested method is called with the corresponding input data, and with an assertion corresponding to the specified oracle.

We rely on JavaPoet[4] to implement the required code generator. JavaPoet simplifies the creation of common JUnit structures, such as test methods, through a directly manipulation of Java constructs as first-class elements, while still being able to insert fragments of Java programs as raw code. The library also allows for the integration of annotations, ensuring proper configuration of JUnit methods and test files, and supports static imports, reducing verbosity by incorporating frequently used assertion methods directly.

Listing 8 shows an excerpt of our code generator. The shown operation produces the code that calls the tested method, and that asserts the correctness of the returned value (i.e., this is when the CTC model includes a `ReturnOracle`). The JavaPoet operation `addStatement` is first used to create the method call, and then used to create the `assertEquals` assertion. `printGeneratedTestData`—not shown due to space limitation—is dedicated to transforming raw input data into valid Java expression that can be passed as arguments. Similarly, `getExpectedValueAsJavaCode` is dedicated to transforming the expected value of a `ReturnOracle` into a Java expression for the assertion.

Based on the oracle of the CTC, there are three main cases managed by the code generator:

(1) In the case of a `ReturnOracle`, the `assertEquals` assertion is used to verify that the returned valued is correct. As an example,

[4] https://github.com/square/javapoet

```java
@Test
@DisplayName("[(WordIsNull, false), (WordIsEmpty, false),
        (WordHasSpecialCharacter, false), (WordHasDigit, false),
        (WordIsPalindrome, true)]")
void testIsPalindrome1() throws IllegalArgumentException {
    actualValue = isPalindrome("baaaa");
    assertEquals(false, actualValue);
}
```

Listing 9: Generated test script for CTC3, i.e., valid word that is not a palindrome (case with a `ReturnOracle`)

```java
@Test
@DisplayName("[(WordIsNull, false), (WordIsEmpty, false),
        (WordHasSpecialCharacter, false), (WordHasDigit, true),
        (WordIsPalindrome, true)]")
void testIsPalindrome2() {
    assertThrows(IllegalArgumentException.class,
        () -> isPalindrome("a0a0a"));
}
```

Listing 10: Generated test script for CTC4, i.e., invalid palindrome due to digits (case with an `ExceptionOracle`)

Listing 9 shows the test method produced for CTC3, where a word that is not a palindrome should produce the value `false`.

(2) In the case of a `ExceptionOracle`, the `assertThrows` assertion is used to verify that the correct exception is thrown. As an example, Listing 10 shows the test method produced for CTC4, where a palindrome word with digits should trigger an `IllegalArgumentException`.

(3) In cases where the CTC lacks an oracle, the generated test script requires manual definition of the oracle. In this case the comment `"// TODO: add oracle here"` is produced, and the software tester should provide an oracle manually.

Once all the concrete test cases are converted into JUnit scripts test, they are consolidated into a Java file. This file is automatically generated and placed in the designated location within the SUT.

3.5 Implementation

We implemented in Java a prototype of the presented approach in the form of a Maven plugin named LNS-TestCrafter[5]. This plugin is centered around a custom annotation named `@GenerateTest`, which must be placed on the Java method to be tested. This annotation signals the plugin to generate tests for the specified method, provided the following list of arguments:

- `PICTInputName`: Path of the PICT combinatorial model.
- `PICTPath`: Path of the PICT executable. Default is empty, and the executable is found in the system `PATH`.
- `PICTAdditionalArgs`: Additional arguments for PICT, e.g., to select a coverage criterion. Default is empty.
- `generatorPackage`: Name of the Java package containing the data generator.
- `generatorMethod`: Name of the static Java method in the `generatorPackage` to use for data generation.

[5]https://gitlab.univ-nantes.fr/E19B907G/lns-testcrafter

```java
    @GenerateTest(
            PICTInputName = "palindrome_model.pict",
            generatorPackage =
                "data.generation.PalindromeDataGenerator",
            generatorMethod = "generateData")
    public static boolean isPalindrome(String word)
        throws IllegalArgumentException {
        // ...
    }
```

Listing 11: `@GenerateTest` annotation applied to `isPalindrome`

- `dirOut`: Path in the SUT folder for outputting generated test files. Default is `src/test/java`.

Listing 11 shows an example of annotation use for `isPalindrome`. When the plugin is called, it discovers all annotated methods, and for each found method applies automatically the complete process described in the paper, starting with calling the PICT executable to produce abstract test cases, and ending with generated test scripts. The Eclipse Modeling Framework (EMF) is used for model management and, as already mentioned, JavaPoet is used for code generation. The provided data generator is called using Java reflexive features.

4 Experiments and Results

This section presents our initial evaluation of the proposed approach on a set of five different Java methods.

4.1 Research question

Two requirements of the proposed approach are the implementation of a data generator for each Java method under test, and extending each combinatorial model with an oracle definition, both of which coming with a certain cost. Yet, if the combinatorial complexity of the test problem is high—and thus if the amount of test cases that are necessary to cover the specification is high—, the "return on investment" of a data generator may exceed its cost. Therefore, this evaluation is centered on the following research question:

RQ: Given a Java method, a combinatorial model, and a target coverage criterion, how does the combined effort of implementing a data generator and extending the combinatorial model with an oracle definition compare to the effort of implementing test scripts manually? And how does this comparison vary with combinatorial complexity, e.g., when changing the target coverage criterion?

4.2 Experimental setup

The presented approach is compatible with static Java methods with primitive input parameters and return values (i.e., strings, integers, booleans and enumerations). Accordingly, in addition to the `isPalindrome` Java method already introduced in Section 2 and studied thoroughly throughout the paper, we selected, adapted and implemented four existing case studies from the field of software testing that fit these requirements:

- `getTriangleType` determines the type of a triangle based on the lengths of its sides. It accepts three integer arguments, and returns either `EQUILATERAL`, `ISOSCELES` or `SCALENE`. An `InvalidTriangleException` is thrown for negative lengths or lengths

that construct an invalid triangle. This method is inspired by an example used in Section 6.1 of the software testing book from Ammann and Offutt [1].
- `getDaysInMonth` returns the number of days in a given month. It takes two integer arguments `month` and `year`, and returns the number of days or `-1` for invalid input (i.e., month outside the 1–12 range, or month outside the 2000–2100 range). The method is inspired by `lengthOfMonth` from the Java standard library[6].
- `findCommand` finds occurrences of a string in a text from a given position in the text. It takes five arguments: the text (string), the string to search (string), whether to match case (boolean), whether to search forward or backward (boolean) and the start position (integer). It returns a list of positions where the string is found. The method is taken from an example used in Section 5.3 of the book of Kuhn et al. [8],
- `validatePasswordStrength` evaluates the strength of a password, based on the amount of upper/lower case characters, special characters, digits, length, and so on. It takes a candidate password as argument (string), and returns either `WEAK`, `MEDIUM`, or `STRONG`. An `IllegalArgumentException` is thrown for null or empty passwords. The method is inspired by the Section 2.3.2 of the work of Yazdi et al. [13].

Then, for each considered Java Method:

(1) A PICT combinatorial model was manually realized based on the specification of the method. This model was then extended with an oracle definition.
(2) A data generator was manually implemented for the method using the Choco-Solver constraint solving library.
(3) The prototype implementing the presented approach was used two times on the method, first using the pairwise coverage criteria, then the 3-wise coverage criteria. Each run yielded a JUnit test script for the method, with a different amount of test cases depending on the coverage criteria.

We used the Lines of Code (LOC) metric to measure the amount of effort required for a given task[7], i.e., we measured the amount of lines of code found in the program obtained from said task. To measure these LOC, we used the tool *cloc*[8], which is able to count so-called *physical* lines (i.e., excluding blank lines and comment lines) of source code in many programming languages.

To answer the stated RQ, for each considered experimental case, we measured the LOC for the oracle definition in the combinatorial model, the LOC of the data generator, and the LOC of the generated test scripts. This last measurement is used both to know how much was automatically generated by the approach, and to estimate the effort that would have been necessary to manually implement these test cases without relying on the proposed approach.

4.3 Results

Table 4 presents the obtained results, and Figure 3 shows a bar plot representation to better visualize and compare amounts of LOC. For the three simpler methods—`isPalindrome`, `getTriangleType`, and `getDaysInMonth`— we observe that the combined effort of both defining the oracle in the combinatorial model (1) and implementing a data generator (2) is higher than the test scripts LOC, both when targeting the pairwise and the 3-wise coverage criteria. In other words, effort-wise, the proposed approach is not "breaking even" as compared to manual work in these very simple cases, most likely because their combinatorial complexity remains low even when targeting 3-wise coverage (maximum of 10 test cases).

For the `findCommand` method, the effort required for using the approach is quite important (161 LOC) due to the rather complex data generator required. The consequence is that this effort is slightly higher than the effort to directly write test scripts when targeting pairwise coverage (149 LOC). However, when targeting 3-wise coverage, the effort required to implement test cases is much more important (436 LOC), making the approach significantly more cost-effective. Yet, note that due to a complex return value, the oracle definition for this case is limited to `"undefined"`, which means oracles must be manually implemented in all generated test scripts.

Finally, the `validatePasswordStrength` method requires a simpler data generator than `findCommand`, but has a much higher combinatorial complexity. This results in the amount of effort required for the approach (81 LOC) being lower both than the effort required to write test scripts in the pairwise case (140 LOC) and in the 3-wise case (608 LOC). Moreover, oracles can be automatically generated and require no manual changes in the generated test scripts.

Analysis and discussion. These results reveal nuanced insights. While simpler cases can achieve satisfactory coverage through automated generation, the overhead of extending PICT combinatorial models and data generators can be higher than manual creation in simpler cases. However, for complex cases requiring a substantial volume of tests, automation proves cost-efficient in achieving comprehensive coverage. In summary, an answer the investigated research question is that the proposed approach requires less effort than manual test scripts implementation *when the combinatorial complexity is high enough*, i.e., when it is worth the effort.

Yet, it can be noted that if the combinatorial model evolves, or even, to some extent, the specification, then it is possible that most changes would need to be made in the combinatorial model, with few changes required in the data generator. Thus, this approach can become interesting even for problems like `isPalindrome` or `getTriangleType`, due to this generative aspect. It can also be argued that the techniques and libraries used for implementing data generators is crucial for the cost-effectiveness of the approach, and that improvements may be possible in that regard.

5 Related Work

To our knowledge, there is little work on how to achieve automated abstract test case concretization in combinatorial testing. It is an explicit step in the combinatorial testing process proposed by Grindal et al. [5] (under the name *translation table*), and vaguely mentioned a few times in the seminal work of Kuhn et al. [8], but without any provided method or automatization in either case. The problem has also been formalized in different contributions [2, 7, 12], but again without any concrete way to generate input data nor test scripts.

Other approaches and tools aim at generating concrete test cases without relying on combinatorial testing at all. Two well-known

[6] https://docs.oracle.com/en/java/javase/17/docs//api/java.base/java/time/YearMonth.html#lengthOfMonth()

[7] While LOC is not perfect as a metric for measuring effort, as two programs of the same size may require very different amounts of development time, we consider this metric a useful and valuable estimate for an initial evaluation.

[8] https://github.com/AlDanial/cloc

Case Study	Oracle Definition LOC in the PICT Model (1)	Choco-Solver Data Generator LOC (2)	(1) + (2)	Generated Tests			
				Pairwise		3-Wise	
				Test Cases Count	Test Scripts LOC (3)	Test Cases Count	Test Scripts LOC (4)
`isPalindrome`	10	66	76	6	41	10	56
`getTriangleType`	5	63	68	7	48	10	63
`getDaysInMonth`	10	62	72	10	68	10	68
`findCommand`	1	160	161	20	149	61	436
`validatePasswordStrength`	13	68	81	22	140	100	608

Table 4: Detailed evaluation results for the considered case studies, in the form of amounts of lines of code (LOC) for combinatorial models, data generators, and generated test scripts for both a pairwise and a 3-wise test suites

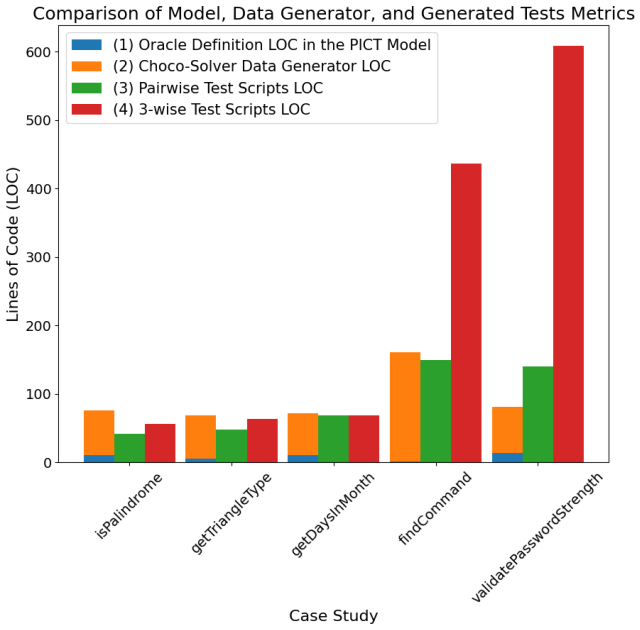

Figure 3: Bar plot of the results shown in Table 4

tools are Randoop[10], which produces unit test scripts by randomly finding valid sequences of statements, and Evosuite[3], which relies on evolutionary algorithms targeting structural coverage criteria (e.g., branch coverage). While this category of approaches can automatically generate concrete unit test scripts with little effort, these approaches do not aim at defining test cases that cover the SUT specification, contrary to combinatorial testing approaches.

6 Conclusion

Concretizing abstract test cases is a required step in combinatorial testing. In this paper, we propose a novel model-driven approach to automate this process. Starting with an extended combinatorial model, abstract test cases are generated and can be concretized using an SUT-specific data generator, Then the resulting concrete test cases can be transformed into executable test scripts using a code generator, making the approach truly end-to-end.

Future research directions include extending evaluation with more sophisticated models and complex input data, such as non-primitive types and nested objects. Additionally, experimenting with AI-based data generators could broaden domain coverage and improve software flexibility.

Acknowledgments

The initial version of this work was accomplished with the consequential help of Naila TINSALHI and Sabrina BENBOUDJEMAA.

References

[1] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing, Second Edition.* Cambridge University Press.
[2] M. Balcer, W. Hasling, and T. Ostrand. 1989. Automatic generation of test scripts from formal test specifications. In *Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification - TAV3 (TAV3).* ACM Press. https://doi.org/10.1145/75308.75332
[3] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE'11).* ACM. https://doi.org/10.1145/2025113.2025179
[4] Thom Frühwirth and Slim Abdennadher. 2003. *Essentials of Constraint Programming.* Springer Science & Business Media.
[5] Mats Grindal and Jeff Offutt. 2007. Input Parameter Modeling for Combination Strategies. In *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering.* ACTA Press, 255–260.
[6] Sebastian Krings, Joshua Schmidt, Patrick Skowronek, Jannik Dunkelau, and Dierk Ehmke. 2020. Towards Constraint Logic Programming over Strings for Test Data Generation. In *Declarative Programming and Knowledge Management,* Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen, and Dietmar Seipel (Eds.). Springer International Publishing, Cham, 139–159.
[7] Peter M. Kruse. 2016. Test Oracles and Test Script Generation in Combinatorial Testing. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* https://doi.org/10.1109/ICSTW.2016.11
[8] D Richard Kuhn, Raghu N Kacker, and Yu Lei. 2013. *Introduction to combinatorial testing.* CRC press.
[9] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2007. IPOG: A General Strategy for T-Way Software Testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07).* 549–556. https://doi.org/10.1109/ECBS.2007.47
[10] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA07).* ACM. https://doi.org/10.1145/1297846.1297902
[11] Charles Prud'homme and Jean-Guillaume Fages. 2022. Choco-solver: A Java Library for Constraint Programming. *Journal of Open Source Software* 7, 78 (2022), 4708. https://doi.org/10.21105/joss.04708
[12] Maria Spichkova and Anna Zamansky. 2016. A Human-Centred Framework for Combinatorial Test Design. In *International Conference on Evaluation of Novel Approaches to Software Engineering,* Vol. 2. SCITEPRESS, 228–233.
[13] Shiva Houshmand Yazdi. 2011. *Analyzing Password Strength & Efficient Password Cracking.* Ph. D. Dissertation. Florida State University.