Informatics

Input Models for Combinatorial Testing using Source Annotations

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Markus Fugger, BSc
Matrikelnummer 01225657

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Mitwirkung: Priv.-Doz. Dr. Dimitris E. Simos

Wien, 28. September 2023

Markus Fugger

Edgar Weippl

Informatics

Input Models for Combinatorial Testing using Source Annotations

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Markus Fugger, BSc
Registration Number 01225657

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Assistance: Priv.-Doz. Dr. Dimitris E. Simos

Vienna, 28th September, 2023

Markus Fugger Edgar Weippl

Erklärung zur Verfassung der Arbeit

Markus Fugger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 28. September 2023

Markus Fugger

Danksagung

Danke an meine fantastischen Eltern, unterstützende Freundin, und die herausragende Universität für ihre bedingungslose Liebe, Unterstützung, und Bildung. Ich bin ihnen zutiefst dankbar für ihre Rolle bei der Gestaltung meiner Reise und meines Erfolgs.

Acknowledgements

Thanks to my amazing parents, supportive girlfriend, outstanding university for their unwavering love, guidance, and education. I am forever grateful for their role in shaping my journey and success.

Kurzfassung

Kombinatorisches Testen (KT) ist eine modellbasierte Testmethodik, die Garantien für die Abdeckung des Eingaberaums bietet. Sie ist äußerst effizient bei der Suche nach Interaktionsfehlern und stellt unter bestimmten Annahmen sicher, dass alle durch eine Kombination von Parametern verursachten Fehler gefunden werden können. Trotz der vielen Vorteile hat sich das kombinatorische Testen noch nicht im Mainstream der Softwareentwicklung etabliert. Der aktuelle Prozess zur Erstellung des Eingabemodells ist komplex und teuer, was zu einer Trennung zwischen Eingabemodell/Testfällen und dem Ziel-Software-System führt und letztendlich zu einem unnatürlichen Änderungsprozess im späteren Verlauf des Entwicklungslebenszyklus führt. Diese Arbeit zielt darauf ab, dieses Problem zu lösen, indem sie ein Proof of Concept (PoC) bereitstellt, das es Entwicklern ermöglicht, ein Eingabeparametermodell innerhalb ihrer eigenen Codebasis zu definieren. Neben der Vereinfachung des Einrichtungsprozesses, indem sie es dem Entwickler ermöglicht, in einer vertrauten Umgebung zu bleiben, automatisiert der PoC auch Schritte, um den Entwickler bei der Erstellung schneller und effizienter KT-Tests zu unterstützen. Die Verbindung zwischen Quellcode und Eingabeparametermodell stellt auch sicher, dass der Änderungsprozess später natürlicher verläuft. Eine Bewertung gegenüber einem externen System zeigt, dass dieser PoC trotz einiger Einschränkungen in der Lage ist, sinnvolle Eingabeparametermodelle und kombinatorische Testfälle sowie einfache Orakel für jeden Test zu definieren. Es handelt sich um die erste Arbeit, die darauf abzielt, Quellcode und Definitionen des Eingabeparametermodells zu kombinieren, um den initialen Einrichtungsprozess sowie den Änderungsprozess des kombinatorischen Testens zu verbessern.

Abstract

Combinatorial Testing (CT) is a model-based testing methodology that offers guarantees about the input space coverage. It is extremely efficient in finding interaction faults, and under certain assumptions, ensures that all faults caused by a combination of parameters can be found. Despite the many upsides, combinatorial testing has yet to make an emergence in mainstream software development. The current input model creation process is complex and expensive, which leads to a disconnect between input model/test cases and the targeted software system, and ultimately to an unnatural change process later down the development lifecycle. This work aims to solve this issue by providing a proof of concept (PoC) which allows developers to define an input parameter model inside their own code base. Besides easing the setup process by enabling the developer to stay inside a known environment, the PoC also automates steps to further support the developer in creating fast and efficient CT tests. The connection between source code and input parameter model also ensures that the change process later on is more natural. An evaluation against an external system shows that, beside some limitations, this library is fully capable of defining meaningful input parameter models and combinatorial test cases, as well as simple oracles for each test. It is the first work that aims to combine source code and input parameter model definitions to enhance the initial setup process as well as the change process of combinatorial testing.

Contents

CHAPTER 1

Introduction

Testing is an integral part of a modern software development lifecycle (SDLC), since it ensures a piece of software exhibits correct behavior in multiple scenarios. Typically, some sort of input is defined, commonly called a test case, which is then sent to a system to either ensure the implementation conforms to a specification (called conformance testing) or to identify potential vulnerabilities (called security testing) [GLL+21].

There exists a multitude of different testing approaches, each with their own unique set of upsides and downsides, which are often tied to various amounts of effort required. Most of the time effort is tied to exploration of input space, which results in varying capabilities of identifying faults or vulnerabilities. One example would be unit testing[AO16], which is very popular among developers since it consumes little time to create, but it also typically only explores a tiny fraction of the input space.

An example which explores more of the input space would be fuzz testing[AO16] where some valid input is chosen as a starting point and then iteratively mutated and submitted as a test case. Since mutation has a high risk of creating invalid inputs, fuzz testing is very popular in the field of security testing, but the input space coverage is probabilistic, which means there is no guarantee that even trivial faults will be discovered with this testing method.

Our last example which is also the focus of this work is model-based testing, in particular combinatorial testing (CT) (more information in Section 2.1). Although it has the biggest time commitment of the three examples, it also offers guarantees about the input space coverage. In CT, the tester chooses a strength t and under certain assumptions it guarantees that all faults which can be triggered by a combination of t parameters or less will be found.

While CT has found its place in some specific areas (e.g. aerospace) it has yet to make an emergence in the mainstream world of software development. One of the biggest downsides of CT is the disconnect of the input model/test cases and the targeted software system. While unit tests only explore a small fraction of the input space, they are directly connected to the source code of a software system, resulting in an easy creation phase and a natural progression when changes occur inside the system. In CT due to the disconnect there is a lot of initial effort in creating an input parameter model (IPM), which is then used to generate test cases which then need to be translated and integrated in the targeted software system. Additionally, changes in source code do not naturally lead to changes in the IPM, often resulting in a disconnect between the IPM and the actual system.

To prevent the disconnect between software systems and IPM, this work introduces methods to define IPMs and CT tests via annotations, declare methods under test and define/provide simple oracle functions. This enables the developer to stay inside his own code base and also eases the initial setup process since he is confronted with known concepts (annotations). Due to the existence of a connection, it is also more natural to change the IPM when source code changes occur. All these methods are encapsulated in a library that also automates some steps such as generating the test set and translating it to a usable data structure and generating executable tests and automatically calling and evaluating the oracle function. Lastly, the library offers an easy approach to integrate it into an existing Continuous Integration/Continuous Deployment (CI/CD) process. Generation and/or testing steps can be executed during a pipeline, which enables the developer to continuously use CT tests during development and deployment. Additionally, this work strives to answer the following research questions:

1. How can Kotlin annotations be used to specify an input parameter model (IPM) for combinatorial testing? These annotations must be capable of defining the available parameters, their domains (value types and ranges), and constraints between parameters. (answered in Section 4.2)

2. How can combinatorial test suites be generated from existing Kotlin sources with annotations and executed in the context of continuous integration? (answered in Section 4.6)

3. What methods exist to define Systems under Test (SUTs) for combinatorial testing in Kotlin, both in the context of native objects (classes and methods) and web interfaces? (answered in Section 3)

4. What requirements exist for functions to be used as simple oracles and how can such oracles be used in a combinatorial testing cycle based on source code annotations? (answered in Section 4.2)

Thus, this work follows the following structure. First Chapter 2 gives an overview and common understanding of combinatorial testing and annotation processors. Afterwards Chapter 3 explores papers tackling similar problems and their solutions. Next Chapter 4 explains the resulting library including challenges, solutions and limitations. Additionally, Chapter 5 explores some of the topics of the previous chapter more in depth on the technical side. Before drawing a conclusion in Chapter 7, the evaluation of the library against an external system is done in Chapter 6.

CHAPTER 2

Preliminaries

This chapter provides a basic understanding of the two core topics of this thesis, namely combinatorial testing and annotation processors. Section 2.1 formally defines CT, explains the general testing workflow, input models and test set generation and lastly talks about limitations of CT. Section 2.2 first gives an introduction to annotations, explaining motivation, use cases, limitations and afterwards gives an overview over annotation processors.

2.1 Combinatorial Testing

To properly introduce CT, we first need to explain some basic definitions.
A step many testing processes share is submitting test cases to the system under test (SUT) [AO16, KBD$^+$15]. SUT can be defined as "a complete system that comprises hardware, software, and connectivity components"[Raa], which is targeted for testing purposes. In the specific case of CT the SUT has a variable amount of inputs callled parameters. Each parameter has a list of potential values it can assume. Both parameters and potential values are described in an input parameter model.

One test case describes the input to the SUT, by assigning each parameter to a specific value. In practice there is often a translation process which transforms the given abstract values of a test case to a concrete file, protocol message or other unit for data transmission, but for simplicity this is not taken into account for the definition. One invocation of the testing process contains a set of test cases, which is called a test set.
One of the defining properties of CT described later on is a claim regarding the portion of the SUT's input space that is covered by the test set. The input space is the totality of all possible inputs which can be submitted to the SUT, defined by the input parameter model. As a consequence this also describes all valid test cases for the given SUT.

When a test case is submitted, an oracle decides if the SUT behaved correctly. The definition for correct behavior is very diverse, since it depends on the SUT and its available outputs as well as the overall goal of the testing process. The goal of traditional CT is to detect faults, especially interaction faults. Faults are instances where either the output or behavior of the SUT deviates from predefined rules. An example would be a system which concatenates two texts. Such a system should never output a text which is shorter than the sum of both input texts or should not crash when big texts are submitted.

On the other hand the goal of security-focused Combinatorial Security Testing (CST) [GLL+21] is to identify vulnerabilities. In this variant the input model is not necessarily based on values accepted by the SUT, but rather on control characters, known token values, or other crafted input designed to modify the way input is processed by the SUT. In the case of language theoretic security [SPBL13], the input model is constructed with the aim of causing weird system behavior which results in an observable effect in the SUT. A big difference between traditional CT and security-focused CST is that there needs to be a deep understanding of how the input will be translated inside the SUT. While both types rely on completeness of the IPM to properly test a system, CST has to also take into account various properties of the underlying SUT such as protocols or file formats to be effective in exploiting a system.

2.1.1 Formal Definition

Since there is now a common knowledge base, it is possible to move on to a more precise definition of CT.
Combinatorial testing is a model-based testing methodology, which takes a model that describes the input space of a SUT as its input and generates a test set which has guaranteed mathematical input space coverage [KBD+15]. The notion of coverage is based on parameter-value combinations. Assume there exists a SUT which accepts inputs made up of $k > 1$ parameters and each parameter with an index i can take values in the range of $\{1, ..., v_i\}$. Observing a selection of parameters together (e.g. indices 2,4,5), we can derive that there are $v_2 * v_4 * v_5$ different parameter-value combinations which can be assigned to the three parameters as part of a test case. Generally speaking if we select t different parameters in a set $S = \{p_1, ..., p_t\}$, where each p_i is the index of a parameter, we can derive that there are $\prod_{i \epsilon S} v_i$ different parameter-value combinations. This combination $(p_1, V_[1m), ..., (p_t, V_{tm})$, where V_i is the set of values which can be assumed by parameter i and m is an index for a parameter value which does not have to be identical for each p_i, is called a t-tuple.

The supporting mathematical structure behind practical combinatorial testing is a multi-level covering array(MCA) $MCA(N; t, k, \{v_i, ..., v_k\})$. This defines an array with N rows, where each row represents a single test case, and k columns, where each represents one parameter. The defining property of a MCA is that for every possible selection of t distinct parameters, all possible parameter-value combinations occur at least once in the array. Naturally choosing a larger t leads to a larger array, since more parameter-value

combinations need to be covered, but also results in better fault detection capabilities since more of the input space is covered. The chosen t is called the strength of a covering array (CA).
To briefly explain the difference between CA and MCA, formally CA is defined as $CA(N; t, k, v)$ which can also be described as a MCA with uniform alphabet, meaning $v_i = ... = v_k$. In practical combinatorial testing this is almost never the case, as such for brevity there is no distinction between these objects in this work, except where explicitly noted. Both variants are called CA.

The above definition leaves the question why there is a need to define the strength of a MCA and not simply setting $t = k$ to always guarantee full exhaustive coverage of the input space. The test set would cover all parameter-value combinations and as long as the input model and oracle are correct it would guarantee to identify all faults of the SUT. These upsides are negated by the sheer amount of test cases this would produce, which is infeasible to execute in practice. To give an example we assume a SUT with 10 parameters, where each parameter can take 10 different values. We know that there are $\prod_{i=1}^{k} v_i$ test cases in such a set, or v^k in the case of an uniform alphabet. This results in an exhaustive test set containing $10^{10} = 10,000,000,000$ tests, which is a staggering amount in almost all practical settings. If we assume a test rate of 1000 tests per second, one run of the test set would take 115 days. Comparing this run time to a test set with strength $t = 6$, this would lead to a test set containing $1,494,326$ tests[Col08], which is a 99.985% reduction and assuming the same test rate results in a run time of less than 25 minutes.

An immediate question arising is how much worse this test set performs in terms of fault detection. Intuitively drastically reducing the size of the test set should also worsen the detection capability by a fair amount. NIST conducted multiple studies regarding known bugs across different industries, in which they were unable to find interaction faults consisting of more than 6 parameters [KLK08]. Some industries had no interaction fault with more than 4 parameters. This suggests that for practical applications the strength can be rather low (e.g. 4-6) to identify nearly all if not all bugs.

It should be noted that this differs in security testing. The input model in CST is typically modeled through an exploit grammar [SZL19], which is used to describe the structure of inputs, which are capable of triggering certain vulnerabilities. The strength in CST increases the proportion of the language described by the grammar that is submitted to the SUT. In contrast to CT, results suggest that higher strength in CST leads to better results and capabilities[SZL19, LGS21, GLL+21].

2.1.2 Testing Workflow

Figure 2.1 shows an overview of a typical combinatorial testing workflow, as seen applied in these works [SGZL19, GO07]. It exists of the following steps:

1. **Input Modeling** As the title suggests the input model is created in this step. All

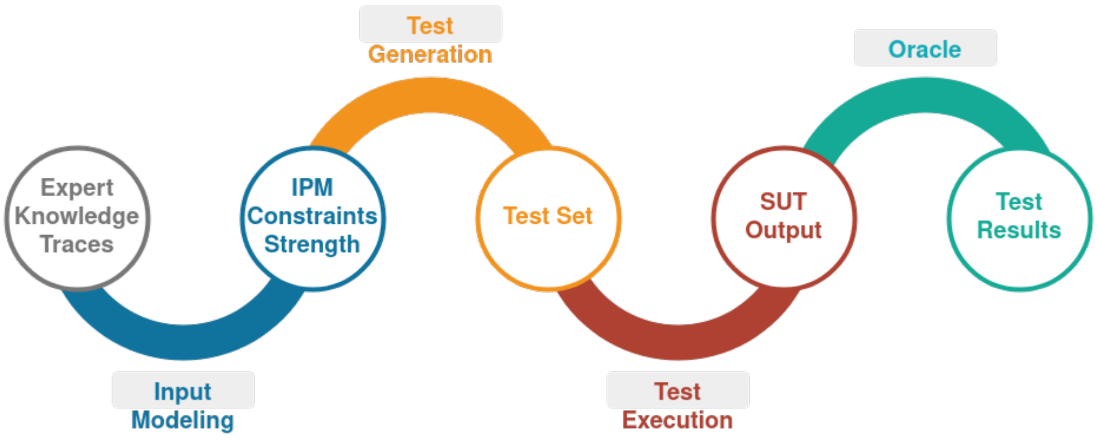


Figure 2.1: Combinatorial Testing Workflow

structures and inputs which are accepted by the SUT are described and additional constraints, which limits the input space and excludes invalid test cases, can be defined. Most of the time this task is performed manually on the basis of expert knowledge or interviews. Typically these interviews are conducted far away from any code base, which especially later on when changes to the input model arise can lead to issues. Thus one focus of my work is to move this definition inside the code base to provide a connection between code base and IPM.

2. **Test Generation** This step combines IPM, constraints and a choice for parameter t and provides it as input for a CA generation algorithm. The purpose of the algorithm is to create a CA with roughly minimal numbers of rows in acceptable time. The resulting array typically consists of rows of abstract values, represented using integers[WKS+20c]. In most cases the values need to be translated into some kind of data structure, file or some other concept to enable test execution. Since this work aims to automate large parts of the workflow, automating this CA generation and translation is also a focus of this work.

3. **Test Execution** After translating the test cases the next step is to execute them. In this process there are numerous factors to take into account. First the test case has to be executed in a way that it is compatible with the specific SUT. Second tests should always be independent and repeatable, which is relevant when facing stateful SUTs. If not taken into account, repeating test cases could suddenly fail or test cases could influence each other. As such most of the time some kind of reset mechanism is used or additional knowledge of the internal state of a SUT is neede to ensure above properties. Most of the time this step is performed with the help of a test execution framework, which monitors the SUT, performs startup and shutdown tasks, triggers potential reset mechanisms, submits the test cases and collects the SUT's output. The form of the output may vary and could consist of

text, function return values or other observable action taken by the SUT, which can be used for the oracle. Since the goal of the above step also applies in this step, automatic test execution is also part of the focus of this work.

4. **Oracle** In the last step the oracle decides if a test case was successful or not. To perform this decision process it uses the output of the SUT as well as potential additional sources such as IPM, test case as well as tables or functions to look up or compute expected result. These oracles are currently constructed the same way as an IPM in the first step, by using expert knowledge and manual work. As such oracles have the same issue of being decoupled from the underlying SUT, which results in high efforts and low adoption rates of CT upon changes in the underlying system. In contrast, for CST, more universal oracles can often be constructed, enabling the ability to use the same oracle for multiple SUTs. This is possible due to the focus of CST on vulnerabilities where exploitation of the vulnerability in multiple SUT's often leads to similar behaviors. As an example the authors of a work revolving around constructing cross-site-scripting(XSS) attack vectors[GRG+19], attempted to execute the alert function. As such the oracle simply had to check if the function was executed to decide if the system was exploited. This simple oracle needs to significant modification to be applicable to new SUT's. In contrast for traditional combinatorial testing a new oracle has to be constructed for each SUT. Only exclusions are simple oracles such as a crash oracle, which deems a test successful if it did not crash, or workarounds such as differential testing[KS17].

2.1.3 Input Model

Input models (IPMs) [KP11] and constraints [YDL+15, CDS07, WKS+20c] often have very similar abstract notations in literature. In stark contrast practical CA generation tools often have significant differences in terms of expected input format, which can be seen in tools such as CAgen[WKS+20c], ACTS[YLKK13], CTWedge[GR18b] or PICT[Cze06]. Different tools tend to focus on different areas of the generation process which partly contributes to the differences in format. As an example, the focus of CAgen is speed, which comes at the cost of supporting less features, such as extension of existing test sets, which is supported by ACTS, or negative testing and submodels, which is supported by PICT.

However, even common features, such as IPMs or constraints, often expect vastly different input formats in each tool. As an example we can portrait an example model taken from the CTWedge Web-Based Editor and Generator[GR18a], which describes a possible configuration for a generic phone. The IPM consists of the following parameters:

- **emailViewer:** Boolean parameter which describes if a phone offers an email viewer.
- **textLines:** Int parameter which describes how many lines of text a phone can display (25-30).

- **display:** Enum parameter, which describes the display of the phone, which can be black and white or offer 16 or 8 million colors.

The example has one additional constraint, which ensures that if the phone has an emailViewer (meaning $emailViewer == TRUE$), it needs to at least be able to display 28 lines (meaning $textLines >= 28$). The example in CTWedge input format taken from [GR18a] can be seen in Listing 2.1.

Listing 2.1: Phone Example CTWedge

```
Model Phone
Parameters:
  emailViewer : Boolean
  textLines:  [ 25 .. 30 ]
  display : {16MC, 8MC, BW}

Constraints:
 # emailViewer => textLines > 28 #
```

Modeling the same example in the ACTS input format can be seen in Listing 2.2. It has slightly less functionality than the CTWedge counterpart. In particular it lacks ranges for numbers (each number has to be written individually) and shortcuts such as Boolean (each value has to be defined individually).The ACTS input format is roughly comparable to the popular *.ini* format in various Windows applications.

Listing 2.2: Phone Example ACTS

```
[System]
Name: Phone
[Parameter]
emailViewer (boolean): TRUE, FALSE
textLines (int): 25,26,27,28,29,30
display (enum): 16MC, 8MC, BW
[Constraint]
emailViewer = TRUE => textLines > 28
```

CAgen offers a web version[WKS+20b] which is capable of importing ACTS input files, but has less constraint support than the original format. It does not support numerical relations such as < ("less than"), which results in the need to define constraints more explicitly. As an example to define the constraint $textLines > 28$ in the above example, it would need to be converted to $textLines = 29 || textLines = 30$ for CAgen. As an upside it does not require types for every parameter, since it handles parameters per default as enumeration of values.

Besides the web version, CAgen is distributed as command line tool which is typically faster and more versatile[WKS+20c]. Due to the limitation of the command line the input format is more condensed and the strength t which is typically separately defined has to be passed in the same command. The phone example in command line version can be seen in Listing 2.3. The speed and versatility also led to using this command line in this work for test set generation.

Listing 2.3: Phone Example CAgen Command Line

```
fipo-cli -t 3 --instance "emailViewer:TRUE,FALSE;textLines
    :25,26,27,28,29,30;display:16MC,8MC,BW;" -c 'emailViewer = "
    TRUE" => (textLines = "29" || textLines = "30")'
```

PICT seems to offer the biggest variety of options including aliasing, variable relations and sub-models. The phone example in PICT format can be seen in Listing 2.4

Listing 2.4: Phone Example PICT

```
emailViewer: TRUE, FALSE
textLines: 25,26,27,28,29,30
display: 16MC, 8MC, BW
IF [emailViewer] = "TRUE" THEN [textLines] > 28;
```

To conclude the constraint support of these tools is comparable in terms of features, but not identical. In detail, they differ in the following points:

- **CAgen:** supports conjunction, disjunction, negation, implication, equality and "not equal".
- **PICT:** supports the same functionality as CAgen and extends it with inequalities (e.g. <, "less than" operator), which is applicable for numeric types and strings.
- **ACTS:** supports the same functionality as PICT, besides negation, and further extends functionality with arithmetic operations, which are applicable to numeric parameters. Additionally it does not support inequalities for strings.
- **CTWedge:** supports the same functionality as ACTS and extends it with biconditional operator (e.g. $a \leftrightarrow b$).

2.1.4 Test Set Generation

For generation of CAs there exist a wide variety of strategies, each with their own advantages and disadvantages[KP11, KS19]. For simplicity this section will only cover a representative sample of methods and further details can be read in the referenced surveys. AETG [CDKP94, CDFP97] is one of the oldest algorithms for generating CAs and is also the predecessor to multiple algorithms such as density algorithm [BC07, BC09], Sliced AETG [KLS20] and PICT [Cze06].

AETG and its successors construct a CA one test at a time and within each test one parameter at a time. Generally speaking the first parameter is chosen by determining the parameter which has the greatest number of uncovered (i.e. not appearing in the test set) t-tuples. The order of the remaining parameters differs from algorithm to algorithm and could be fixed, random or heuristically determined. The value of each parameter is chosen by maximizing the number of newly covered, currently uncovered t-tuples. This process terminates when no t-tuples remain uncovered.

The original AETG algorithm has been shown to be less efficient than algorithms which take the structure and requirements of a combinatorial test set into account[KS19]. This is due to the fact that the original implementation is more or less a popular greedy set covering algorithm translated into the domain of CAs. The only remaining publicly available and competitive implementation of an AETG-style algorithm is PICT.

Currently the most relevant algorithm in practice is the In-Parameter-Order Algorithm (IPO), which was originally presented in 1998[LT98] and later extended to higher strengths[LKK+07]. Both ACTS[YLKK13] and CAgen [WKS+20c] use IPO as the underlying algorithm while CTWedge[GR18b] offers it as one option. Due to popularity this algorithm has seen multiple extensions and improvements such as including a quantum computing approach[WKS20a], using symmetries in CAs for improvement [CG09] or combining IPO with simulated annealing[WKS21].

At its core IPO is a greedy algorithm which orders parameters descending by parameter size. As a starting point it constructs a test set for the first t parameters and then subsequently repeats following 2 steps:

- **Horizontal extension:** In this step the test set is extended horizontally by adding a column to the test set, one row at a time. The value for each column is chosen from the alphabet of the parameter and maximizes the number of uncovered t-tuples becoming covered. After this step the test set has the same number of rows and one more column. It is important to note that t-tuples relevant for the appended column can still remain uncovered.
- **Vertical extension:** In this step the test set is extended vertically by adding rows until all t-tuples involving the column added in the previous step are covered. This extension can lead to undefined values for some columns called $don'tcare$ values, which can assume all valid values for the respective parameter in the column. Later vertical extensions can still change these $don'tcare$ values to a fixed value. After this step the resulting test set is a CA of strength t for all columns which have been added by the horizontal extension step.

The IPO algorithm terminates after it completes the vertical extension for the last added column. Generally speaking IPO tends to be the fastest algorithm for many problem instances [WKS+20c]. The downside is that most of the time the resulting CA is slightly larger than CAs produced by e.g. simulated annealing.

Simulated annealing, in particular Covering Arrays by Simulated Annealing(CASA)[GCD09, GCD11] also uses a two-step process, containing the *outer search* and *inner search* and are executed in the following way:

- **Outer search** As a first step the outer search tries to identify the smallest number of rows to construct a CA satisfying given strength, parameters and parameter values. It then starts the inner search which executes the simulated annealing process.
  - **Inner Search** Simulated annealing starts with a random array of k columns and some N, which is determined by the current iteration of the outer search. It then determines the coverage of the current array.
  - **Inner Search** Then it randomly replaces an arbitrary cell in the array and recomputes the coverage of the resulting array. If the coverage improved or stayed the same then the new array is accepted as the array which will be further annealed. If the coverage worsened there is a formula which will calculate if the array is still accepted, depending on temperature (a value which decreases over the runtime of the algorithm) and the margin of how much worse the coverage became. This is done to prevent being stuck in local minima/maxima.
  - **Inner Search** The above steps are iterated until there is no improvement for a number of iterations or the array achieved full coverage (becoming a CA). When one of the criterion is met the inner search returns to the outer search.
- **Outer Search** Upon return of the inner search the outer search will adjust the target N and restart the inner search. The outer search terminates when a condition is met, which is typically a timeout.

CASA is also an option for CTWedge and produces slightly smaller CAs than the greedy approach but has a much longer runtime as a trade off.

2.1.5 Limitations

CT and CST offer a guaranteed level of input space coverage or exploit space coverage in CST, while maintaining a relatively small number of required tests. If certain conditions are met it can identify all faults which are triggered by up to t parameters. Unfortunately in practice most of the time not all conditions can be met, thus leading to limitations of the whole CT/CST process. These limitations include:

- **Correctness and Completeness of input parameter model** CT heavily relies on a specification of the input space, which is a typical occurrence in all model-based testing approaches. Unfortunately such models are often not available in practice, since if there is no anticipation of formal methods verifying the system (e.g. in secure development lifecycles), then developers mostly do not invest time in creating and maintaining an input parameter model or similar description. Since this work tries to tackle bridging the gap between input model and developer this is one of the main issues addressed. Additionally for large domains of parameters (e.g. 32-bit integer) it is impossible to test the whole value range which results in the need to apply techniques such as equivalence partitioning[Rei97] to reduce the input domain to a manageable subset. For CST, to my knowledge, there is no practically useful measure of completeness that would be applicable to models of exploits.

- **Correctness of execution environment** CT and CST assume that there is some execution framework which will correctly submit generated test vectors. In this work part of this implication will be handled by the library, since it provides an environment which will automatically submit test cases to a method. Due to limitation of this simple PoC we still offer the possibility of additional setup before execution.

- **Independence of tests** CT and CST assume that the SUT will process each test case individually, which is typically not the case in practice. Often test cases lead to side effects in the system, such as data created in a data structure or leaving the SUT in a particular state. This can change the behavior of following test cases. A rather trivial solution would be the use of a reset symbol[SL89] that restores the same state of the SUT every time, however to my knowledge there is no practical work employing CT or CST using this feature. Due to the simplicity of the library it also assumes that each test case is handled individually.

- **The oracle problem** CT and CST assume there exists an oracle function, which observes the output of the SUT and decides if the test case was successful. Constructing such an oracle function involves a fair amount of challenges and is known as the oracle problem in software testing[BHM+14]. While there are numerous solutions, none seems to offer an approach which has high accuracy and low effort, which would make it ideal for practical software testing.

2.2 Annotation Processors

Annotation Processors, as the name suggests, process annotations, which is why a proper introduction to annotations is needed to provide a common knowledge base for further explanations. Annotations discussed in this section revolve around Kotlin annotations. Kotlin is a programming language which depends and interoperates with Java and fully supports Java annotations. Since both Kotlin and Java annotations fulfill the same purpose and Kotlin annotations are derived from Java annotations, they will be used interchangeably in this section. There are also annotations in other programming languages fulfilling the same purpose, for example in C[HNS09], but they are not the focus of this work.

2.2.1 Annotations

Java introduced annotations in J2SE 5.0 in 2004[GJSB05]. They are used to provide metadata information associated with Java elements i.e. classes or methods. Metadata information is essentially data about data, providing additional information about associated elements in code. Per default annotations have no direct effect on the code they annotate and also do not influence how code is compiled in any way.

In an implementation context annotations can be considered special interfaces[GJSB05, Dar09]. To distinguish an annotation from a normal interface Java uses the @$interface$ keyword, but in Kotlin we use the simple *annotation* keyword instead of $interface$. Annotations can hold an arbitrary amount of additional values, which normally includes the actual metadata, which is called annotation parameters. An example declaration can be seen in Listing 2.5, while Listing 2.6 shows how to use the declared annotation. An annotation declaration can be created anywhere inside the source package of a project. Additionally the location of the declaration also determines the visibility of the annotation inside the project, since depending on the structure of a system it may be visible to the whole project or only a specific part of the system.

Listing 2.5: Annotation Declaration Example Kotlin

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.SOURCE)
@Repeatable
annotation class Example(val metadata: String)
```

Listing 2.6: Annotation Usage Example Kotlin

```
@Example("meta") class Foo {
  @Example("data") fun baz(foo: Int): Int {
      return 1
  }
}
```

Annotations can be divided by three properties: Target, Retention and Repeatability. Target limits which elements can be annotated by declared annotation, for example $AnnotationTarget.CLASS$ limits the annotation to only annotate class elements. Retention configures if annotations are present in binary code and if they can be accessed via reflection, for example *AnnotationRetention*.$BINARY$ leads to annotations present in binary code but not accessible via reflection. Repeatability configures whether the same annotation can be used multiple times on the same element or not and is configured via @*Repeatable* annotation. Java and Kotlin both also have some predefined annotations. Examples include compiler annotations such as @*Override* or meta annotations such as @*Repeatable* [Ora14].

Annotations can be used in many different use cases [GJSB05, Dar09, RV11, ND08] such as:

- **Compiler Information** Developers can use compiler related annotations to interact with the compiler to suppress compiler warnings or detect errors. An example would be the @$SuppressWarnings$ annotation which can be used to suppress a multitude of warnings such as warnings for using a deprecated function.

- **File Generation** In this use case a developer annotates an element with a predefined annotation of an arbitrary annotation processor, which is then picked up by the processor and results in a generated file dependent on the used annotation. Since this is the use case relevant for this paper, an example could be @$CTATest$, which generates a combinatorial test class, which is an annotation of this work.

- **Documentation** With the use of the @*Documented* annotations, developers can signal that annotations with a type should be documented by Javadoc or other tools.

- **Logging, Testing** Annotations are not only relevant during the compile process, but can also be used and processed during runtime, which enables possibilities to use them for logging or testing.

- **API, Libraries** Many APIs or libraries rely on annotation to provide their functionality. As an example, Spring is a library which enables nearly all functionality and configuration via annotations. It is possible to turn features on or off or provide congifurations (e.g. timeout for request) or orchestrate features. Simply annotating a class with @*Entity*, will mark this class as a database entity class and trigger a variety of additional functionality automatically.

While annotations have many usages and are widespread in the Java/Kotlin world they also have some limitations[CV14], which also impacted the implementation of the library in this work. Relevant limitations include:

- **Limited granularity** Although most elements can be annotated, there are also some which cannot, including generic statement, expression or code blocks. This leads to the inability of annotating and adding metadata inside the body of a method.

- **Limited Parameter Types** Annotation Parameters have very strict type limitations. As an example, Java parameter types only include primitive values, strings, enums, classes, other annotations and arrays containing preceding types.

- **Limited Parameter Values** Annotation Parameter Values also have string limitations, since parameters can only hold values resolvable in a static context. This leads to the inability to use any kind of method or string template as value for an annotation.

2.2.2 Processors

The purpose of an annotation processor is to process annotations and provide access to the metadata contained inside them. This metadata is then used to trigger some kind of functionality which also heavily depends on the type of processor. In general processors are divided by time frame they access annotations, either during compile time or during runtime[PN15]. Access during compile time is realized via an annotation processing API, which is also the approach used in this work, while runtime processors use reflection API for access.

While both approaches offer metadata in different representations, in general they are very similar since both approaches offer all available metadata in the relevant processing phase. One of the reasons why metadata can change between compile- and runtime is the retention policy of annotations. As explained above certain retentions prevent annotations from reaching binary code or being accessed by reflection leading to some annotations only available during compile time. Compile time processors also generate code most of the time, which can contain annotations which could also lead to a difference in metadata between compile- and runtime.

These approaches not only differ in provided metadata, but also in capabilities. During runtime a developer can invoke methods both known or unknown during compile time, while compile time processing does not allow invocation of code at all, since the source code is not compiled yet. Another difference is that compile-time processing is only allowed to generate code and not alter existing code, while reflection allows you to alter code i.e. change type or add a constructor.

From an implementation perspective these processor types also operate on completely different data structures and method sets, which can lead to more complexity if developers have to work in both timeframes. This led to multiple approaches to bridge the gap between both processing approaches[PN15, Paw06], but the library used in this work, called ksp[Goo21], still adheres to above limitations.

CHAPTER 3

Related Work

This section about related work is split into two parts. The first part is revolving around papers tackling similar issues than this work in model based testing. The second part takes a look at papers which involve testing in Java and their approach to define systems under test (SUT).

3.1 Issues in Model-Based Testing

As stated in the introduction two of the biggest pain points of model-based testing is the creation of a model, in the specific case of CT an IPM, as well as how to deal with changes after an initial model was created. Consequently there have been multiple papers addressing one of the two issues and offering various solutions to them.

Works revolving around the model creation process try to address the issue that oftentimes no model exists and it takes a lot of effort to create one. A paper revolving around web application testing [BSSH17] argues that model based testing would provide plenty of advantages for it, since a large amount of effort is put into ensuring that users have an uniform user experience across all platforms. This desired experience could be modeled and then used to create uniform tests for all platforms. The authors argue that unfortunately most of the time these web apps have no model or even a specification of desired behavior. To overcome this issue they propose learning based testing, which revolves around defining actions of the system and desired test cases and then using a system, which sends test cases and evaluates the output to create a model of the SUT through learning. The resulting model is then used as the base model and can be further adjusted as needed.

Similar to the above work there is another paper which tries to fix the issue of model creation with exploratory learning [FBJ16]. The focus of this work focuses more on the effort it takes to create a detailed and accurate model, which is needed to leverage all the upsides of model based testing. In general exploratory testing is about manually exploring the possibilities of a graphical user interface (GUI) as a tester. The downside of exploratory testing is that the quality of tests is unclear and heavily depend on the tester as well as there is no assurance that a feature was tested. To circumvent these downsides the authors propose an approach where a system tracks the testing process and offers information and possibilities to mark important steps to the tester. In the background the system attempts to identify all relevant steps of the tester and transform them into a usable model, which could then be used for further model based testing.

In contrast to the previous two papers which involved at least to some extent manual work the next two papers try to create models without any additional manual work required. The first of these works revolves around Javascript libraries[MT19]. The authors explain that Javascript libraries are most of the time regularly updated, but often developers have a hard time to determine if a new update contains breaking changes or not. To support developers in the future the paper proposes to use model based testing to create regression tests, which should ensure that future releases do not contain any unknown breaking changes. This approach works without any prior model, since it heavily depends on existing clients using the library. The system downloads tests of these clients, executes them and tries to build a model by analyzing the interaction between client and library. After the initial model creation this model then can be used to execute regression tests against the new update and reports if any interactions differ from the previous interactions.

The second work revolves around mobile tests [AFT+14]. The approach of this paper is to attempt to reverse engineer the GUI to create a model. It is similar to the exploratory learning approach in the sense that the program which attempts to reverse engineer the app, obtains all possible events and options of the initial starting screen and explores the application via this information, gathering more information with every new screen. The finalized result of the reverse engineering process is a state machine model which could then be used for further test generation.

Although the above papers try to tackle a similar issue as this work they use fundamentally different approaches, which focus more on either reducing or completely eliminating the need for manual work in the model creation phase. In contrast to these approaches which all ignore the issue of how to ensure that changes are reflected correctly after the initial creation as well as containing uncertainties of how complete and usable the resulting models really are, the approach in this work focuses more on breaking up the big chunk of work into smaller pieces and enabling a larger group, namely developers, to work on smaller pieces of the model creation process in parallel by integrating it naturally in the development process.

Works revolving around the issue of how to deal with changes in created models have a really strong focus on how to integrate the modeling as early as possible, since then the model is present at all stages of the development process resulting in a more natural adjustment process due to the reason that the model already constantly evolved. To further improve the quality of the model some works also try to enable additional stakeholders or roles the ability to influence the model. This also provides the upside that more people are incentivised to update the model upon change.

This is exactly what the first paper revolving around model-based testing (MBT) in software systems[EWZC15] does. It proposes a process using usage models to start at an early stage and also involves additional stakeholders and the software team. The idea of the process is that although the initial creation of the usage model is done by an expert quality assurance (QA) person, this usage model is then refined with multiple feedback iteration involving previously mentioned stakeholders and software team. This usage model can also be adjusted along the way and is tied to requirements, which is a reoccurring theme in these types of works. At last this usage model is then used to generate test cases.

The same idea of starting early in the process and involving additional stakeholders can also be seen in another paper revolving around using UML diagrams as models [HGB08]. The authors argue that UML diagrams can be created early on, purely based on requirements and then further adjusted later in the process. Since the diagrams are also created from a user perspective it is easy to understand by stakeholders and experts. These UML diagrams are then used to generate test sets, which can then be further used in tests.

Although the main focus of the next work is more about combinatorially testing 4 systems[DJK$^+$99] and evaluating the performance, the authors also touch upon the issue that current tools expect many testers to be $\frac{1}{3}$ developer, $\frac{1}{3}$ tester and $\frac{1}{3}$ system engineer. These types of people are very rare, which means there needs to be alternatives to spread the possibilities of CT to more QAs, without the need of extensive requirements. The solution this work proposes is using a tool called AETGSpec, which is simple to use and understand, but falls short in providing the same depth as other tools. The authors argue that although the tool has less functionality it still offers the possibility to craft high quality test cases. Besides this decision the rest of the process is a standard manual CT process.

The last work is a more unusual one, since it revolves around voicing and resolving uncertainties in a model[CBGS18]. Instead of creating an initial model and iteratively changing it during the development process, they propose a process in which the system is modeled as complete as possible with given requirements and voicing uncertainties inside the model when aspects are still unclear. Part of the system they proposed then observes the system during test execution of script with uncertainties and reports back to the modeler. This modeler can then adjust the model, remove or add uncertainties and restart the process until all uncertainties are gone.

The papers described above vary a bit more in focus than the creation centered papers. The first two papers [EWZC15, HGB08] try to start modeling as early as possible and involve as many people as possible, but there is still the disconnect of model and system. Additionally both approaches also do not have any automation for translating to usable test cases or providing an automatic process at all. The third paper[DJK$^+$99] only revolves around involving more people and is completely manual, which is the opposite of this work. The last work[CBGS18] had more automation, but it had an entirely different focus than this work, since it revolved more about how to tackle changes in models instead of recognizing an issue due to the disconnect of model and system.

In terms of similarity the most similar work revolves around MBT using two tools to automate a lot of the process[MPS19]. The first tool revolves around connecting requirements with test specifications, which ensure testing is done from the requirements phase and counteracts the issue of starting testing too late. This connected model is then used to generate executable test scripts via the second tool called Robot. The authors argue that the upside of the Robot framework is that it already has a lot of built in functionality which does not need any further implementation to use. Although it is similar in some aspects e.g. creating executable test scripts, there still exists the issue that the model and system are disconnected. Additionally in contrast to this work the process also involves two tools which have to be used manually.

3.2 Define Systems under Test in Java/Kotlin

To answer the research question: "What methods exist to define Systems under Test (SUTs) for combinatorial testing in Kotlin, both in the context of native objects (classes and methods) and web interfaces?" a selection of papers were analyzed which revolved around testing in Java. The reason why Java´ was chosen over Kotlin was to extend the range of papers to choose from, since Kotlin is still a rather new programming language. Additionally papers which revolved around testing with annotations were chosen preferably.

Papers revolving around native objects offer a variety of approaches. One option presented which is also used by JUnit is simply annotating the relevant method to test[RV11]. If the need arises additional information can be provided via annotation parameters. A similar option is annotating a relevant class. This option came in two variations. The first variation is that upon annotation of a class all methods inside it were marked as methods under test and additional annotations could be used to opt-out some unnecessary methods[PE07]. The second variation is that upon annotation of a class all methods inside are potential candidates to opt-in as method under test via a custom annotation[PJ09].

In the context of classes another option has been proposed. This approach views one class as a container which holds all information relevant for a test case, which also includes the method under test, and provides various annotations to mark setup or teardown tasks[BOF14]. This approach was also used in this work, since it enables a lot of flexibility to add more functionality around a test if needed (e.g. setup, teardown). Additionally changes to the method under test are immediately noticed, since not adjusting the container class will lead to compile errors, which is not guaranteed when simply referencing a method via string. The last option does exactly that by referencing a method under test via string, which includes method name, class name and package name[Mar05]. Although this method is very error prone in itself the paper proposing it offers a large variety of ways to define these properties, which is worth mentioning. Ways to define a method under test proposed by the authors includes defining it in a Javadoc/comment, referencing it in an annotation, defining in it an external XML file or using a method which returns the information as a string.

Web interface papers show limited diversity in their approaches to solve the problem. All papers found, which tackled SUTs in an web interface context used a nearly identical approach, namely generating a web client which accesses the web interface for testing[HL05, MBX07a, TPW$^+$02, MBX07b]. This approach defines a SUT by defining possible actions and inputs via client generation which needs some kind of specification and then simply defining an URL to specify the specific web interface. The specification used has been Web Services Description Language (WSDL) document in all cases, which is a way to describe web services. Although an argument can be made that WSDL is an old standard, which can also be seen by the release date of referenced papers, this approach is still used with the only difference that tools and specification languages have changed. Nowadays someone would probably specify a web interface via OpenAPI[Ini11] and generate it via a language specific generator[Col18]. Thus this is still a valid approach and also has some upsides such as changing a web interface simply results in a regeneration of a web client.

CHAPTER 4

Methodology

As stated in section 1, the main purpose of this work is to create a PoC in Kotlin that bridges the gap between IPM and SUT. It automates additional steps of the CT process for a smoother starting point. The idea is to use annotations to define an IPM. The workflow involves the following steps:

1. Define IPM, target SUT and oracle function.
2. Invoke library, which generates tests.
3. Execute generated tests.

This is, of course, a very abstract and basic overview of the workflow since both the first and second steps involve a lot of substeps, which will be explained subsequently in the next sections.

4.1 Overview

To enable a more concrete overview of the workflow of the library, a closer look into each step is needed. The first step consists of several substeps, which can also be seen in Figure 4.2a:

- **Annotate Input Class**: This input class represents the input to the SUT. Developers annotate it to define parameters and constraints.
- **Implement Oracle Function**: Developers implement an oracle function to determine test success. They can create the function before or after the library generates tests.

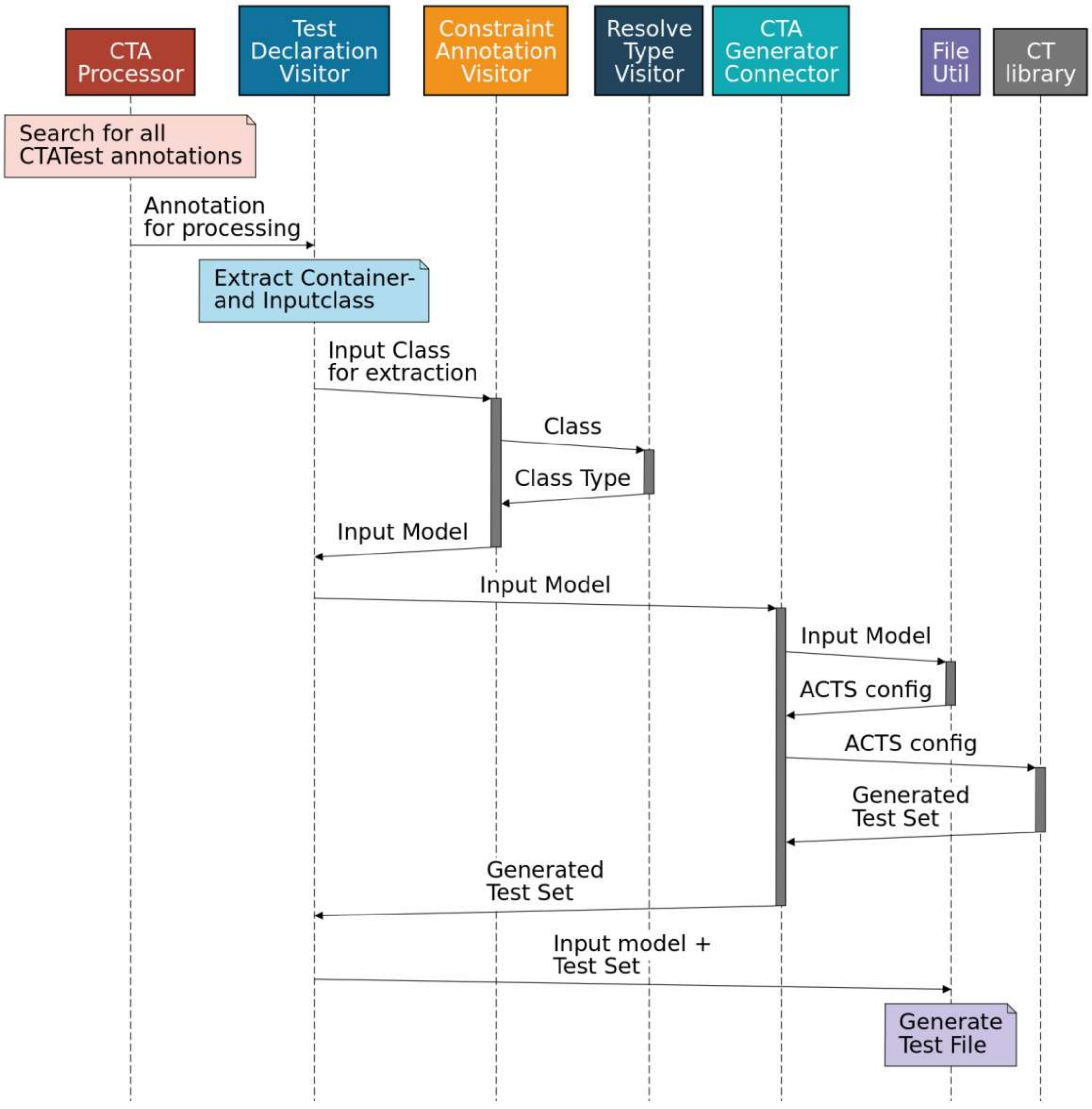
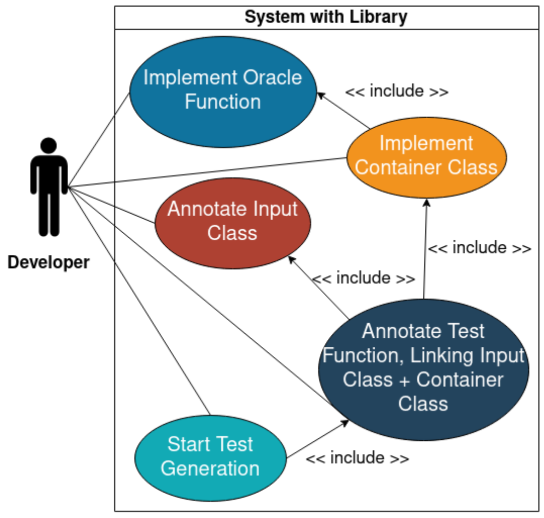


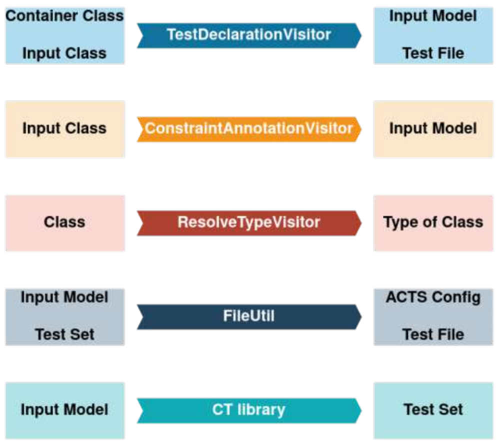
Figure 4.1: Sequence Diagram of library workflow

- **Implement Container Class**: This is a wrapper class that contains the target function to be tested, as well as the oracle function.
- **Annotate Test Function, Linking Input Class and Container Class**: Developers annotate a function that declares a single CT test and references the input and container class.

All these steps lead up to the ability to invoke the library and start the test generation. Most of the above steps rely on annotations. A closer look at annotations will be given



(a) Use case diagram of library



(b) Input/Output diagram of library

Figure 4.2: Use case and Input/Output diagram of library

in the next section. All this setup leads up to the second step, which is invoking the library to generate tests.

This step involves no interaction from any user and once started works completely automated. The only possibility for possible interaction is if the developer made a mistake in a previous step resulting in the need to fix the issue before rerunning the process. This step contains a lot of technical substeps, which will be explained in Sections 4.3 - 4.6.

In general, the workflow can be described as scanning the source code to extract all the information defined during the setup phase and converting that into executable tests. An overview of the components involved and their interaction can be seen in Figure 4.1, while an overview of inputs and outputs of components can be seen in Figure 4.2b. More details will be provided in the following sections.

After the creation phase, the developer can move on to the last step, which is executing the generated test cases. Typically, this step will also involve tweaking of the oracle as well as readjustments of the IPM definition if needed.

4.2 Annotations

This section provides an in-depth look at the annotations used in this library while answering RQ1: "How can Kotlin annotations be used to specify an input parameter model (IPM) for combinatorial testing? These annotations must be capable of defining the available parameters, their domains (value types and ranges), and constraints between parameters."

The first challenge in designing the library was to define an appropriate annotation format that could handle all the requirements. As stated in the research question, the annotations had to be capable of defining parameters and their domain, as well as constraints between parameters. The goal was to implement the library with heavy use of annotations to provide a uniform entry point for all interactions. This led to additional requirements that needed to be considered when defining the format, including:

- Defining a SUT, which is a function in a software system in the context of the PoC.
- Referencing input classes that contain the defined parameters and constraints.
- Referencing an oracle, which is a function in a software system in the context of the PoC.
- Providing a way to define an automated CT test.

Additionally, a way to enable more concrete debug logs to support developers when interacting with the library was deemed a useful addition to the format. The creation of the annotation format was an iterative process. The first step was to create a format for defining parameters and constraints. This step included researching and deciding on a test set generation program, since there are varying capabilities and formats between different tools. For this PoC, CAgen [WKS+20b] as a command-line tool was chosen. A deep dive into how the tool was integrated and challenges involving the process will be given in Section 4.5.

For defining parameters and their domains, CAgen supports enumerations, integer values, and boolean values. As stated previously, this library uses the concept of input classes, which means one class (in a Kotlin context) represents a self-contained part of an input model. This input class only contains parameter definitions and constraints revolving around parameters contained in this class. In the current PoC, only a single input class is supported, but the implementation took the possibility of multiple input classes into account. Especially in the context of functions, the possibility of having multiple inputs is very probable.

The upside of using these input classes is that the parameter definition is done on the actual parameter in the class, which ensures a connection between parameter definition of an IPM and the actual parameter in the software system. A simple example of this concept in pseudocode can be seen in Listing 4.1.

Listing 4.1: Simple example of the concept of annotating parameters

```
/**
* This example contains a normal integer parameter which
* represents a parameter definition inside a class. The
* annotation on top adds metadata to the parameter defining
* its definition inside the IPM. Due to the natur of
* annotations parameter and annotation are
* automatically connected.
**/
@AnnotationDefininingIPMParameter
var actualParameterInClasscontinuous integration = 1
```

The initial concept leads up to the finalized parameter annotation concept, which includes the following annotations:

- **CTABoolean:** This annotation is used to define a boolean parameter. It has no additional parameters, since boolean parameters can only be either true or false.
- **CTAInt:** This annotation is used to define an integer parameter. It has three additional parameters and can be used to define integers in 2 different ways.
  - **Range:** Although as stated above CAgen does not support range integers, the library itself converts the range to a list to enable more flexibility when defining parameters. For this definition 2 parameters are relevant *from* and *to*, which defines the start and the end of the range.
  - **Value List:** When defining integers this way we use the third parameter available which is just a simple list, where we can provide all values the integer can possibly take, which is then simply used for the IPM.
- **CTAEnum:** This annotation is used to define an enum parameter. It has one additional parameter and can be used in 2 different ways.
  - **Enum:** If this annotation is placed on an enum class in Kotlin, no additional parameter needs to be set, since the library itself extracts the enum type and possible values.
  - **String Enum:** To also enable the possibility of string enums (which is a string, which is expected to have a value out of a limited set of values), this annotation is an enum, which is just a simple list which can be used to define all the possible string values the enum can take similar to the list on the integer annotation.

With these annotations, we can define all parameters currently supported by CAGen. The next challenge is constraints between parameters. CAgen currently only supports if constraints, which are logic formulas with restricted support of operations. They are commonly referred to as "if constraints" within this thesis due to their typical form: *IF A THEN B*. Supported operators are $\&\&(AND), ||(OR), =>$ $(IMPLIES), !(NOT), EQUALS(==), NOTEQUALS(!=)$, which are used to define constraints.

To bundle the constraints in one location and prevent them from being spread across multiple locations, a wrapper annotation called **CTAConstraints** was created. It contains a list of if constraint annotations **CTAIfConstraint** as parameters. Initially, the idea was to reference existing parameters in a Kotlin class to ensure verification and prevent errors like typos. However, limitations of annotations changed the design, as annotation parameter values must be compile-time constants and cannot reference any type of function or non-constant value.

Besides enabling the possibility of referencing existing parameters, another idea was to define constraints with building blocks. However, a limitation of annotations is that they cannot self-reference, which creates issues when trying to translate a formal grammar with self-referencing into an annotation format. As a result, the decision was made to use a simple string input for constraints and define some enums for the operators to provide developers with some support. The downside of this approach is that constraints need to be written exactly in the format expected by the test generation tool. An example of defining an IPM using all parameter and constraint annotations can be seen in Listing 4.2.

Listing 4.2: Example showing usage of annotations to define an IPM

```
/**
* simple example enumeration class
* can take values: Value1 or Value2
**/
enum class EnumExample {
  Value1, Value2
}

/**
* constraints annotation containing 2 constraints
*
* first constraint ensures that if the normal enumeration value
* is Value1 the string enumeration value cannot be "RED".
*
* second constraint ensures that if the integer parameter with
* range definition is 2 then the integer parameter with a value
* list is 4.
*//
@CTAConstraints(
  CTAIfConstraint("enum $CTEQUALS \"Value1\" $CTIMPLIES
     stringEnum $CTNOTEQUALS \"RED\""),
  CTAIfConstraint("intRange $CTEQUALS 2 $CTIMPLIES intValueList
     $CTEQUALS 4"),
)
class ExampleClass {
  //integer parameter, defined to be in range 1 - 5
  @CTAInt(from = 0, to = 5)
  var intRange: Int = 1

  //integer parameter, defined with valid values 0,1,2,3,4,5
  @CTAInt(values = [0, 1, 2, 3, 4, 5])
  var intValueList: Int = 3

  //boolean parameter, can be true or false
  @CTABoolean
  var boolean: Boolean = false

  //enumeration value, valid values are defined by enumeration
  //class on top
  @CTAEnum
  var enum: EnumExample = EnumExample.Value1

  //string enumeration value, defined to be either "RED",
  //"GREEN" or "YELLOW"
  @CTAEnum(values = ["RED", "GREEN", "YELLOW"])
  var stringEnum: String = "RED"
}
```

After defining the above format, the next small step is to create an annotation for debugging purposes. A simple annotation called **CTADebug** is created. When this annotation is set on any element, it enables the creation of a debug log file with more information about the scanning and creation service.

The next challenge is to fulfill the additional requirements besides the research question. These requirements include referencing an SUT and input classes, as well as defining a CT test in general. To address this, another annotation called **CTATest** is used. This annotation determines if the library is active and fulfills all the necessary requirements. As soon as a single CTATest annotation is present, the library becomes activated. This annotation also defines a single CT test, which describes an input model and the SUT, and starts the automatic test generation process. The input class and SUT are defined via parameters of the annotation.

There are many libraries out there which provide an annotation, which enables provided functionalities (e.g. @*EnableXXX*). Typically these libraries provide parts of their functionality out of the box, which is not possible in the context of the library, since we need some information to provide any meaningful functionality. That's why we went with the above solution. An example usage of the annotation can be seen in Listing 4.3.

Listing 4.3: Example usage of CTATest

```
@CTADebug
class CTATests {
  @CTATest(CTATestContainerImpl::class, TestClass::class)
  fun testingTestClasses() {}
}
```

The **CTATest** annotation contains 2 parameters: *testContainer*, which references a test container class, and *classToTest*, which references all the classes that should be used as input class for the test. The references to input classes are vararg (variable arguments), which means multiple test classes can be referenced and collected into a list. However, the current version of the library only supports one class. Due to the limited scope of the PoC, we decided to use a simple approach instead of extracting all input classes from a referenced method and working from there towards the input model. The test container class is a specific class that must implement a predefined interface. You can see the interface in Listing 4.4.

Listing 4.4: Interface for container class

```
/**
* Interface for TestContainer.
* A TestContainer holds a reference to the method which will be
    tested as well as the oracle function which will then
* decide if the test was successful.
*/
interface CTATestContainer {

  var testMethod: KFunction<*>

  /**
   * Oracle Function which receives the function call and
      inputs for a singular test run as input and returns the
   * success of the test run as boolean as output.
   * @param methodToTest function call for test run.
   * @param inputs input for test run
   * @return if test run was successful
   */
  fun oracle(methodToTest: KFunction<*>, inputs: Array<Any>):
     Boolean
}
```

The container class serves two purposes: firstly, it enables the definition of a SUT (System Under Test), which, in our case, is the function to be tested. Secondly, it allows the definition of an oracle function. Although it may seem like a strange workaround to create an interface just to reference some methods, this decision was made due to limitations of annotations. Initially, the idea was to directly reference two functions inside the annotation. However, unlike classes, functions are not allowed as parameter values inside an annotation. This led to the need for a different approach.

Some libraries, like Spring JPA, use string identifiers to match annotations. For example, in the context of database one-to-many relations, you would reference the field in the parent class with a simple string identifier. This concept of matching strings led to the idea of annotating the functions with an annotation that has a string parameter. This string parameter is then used to match the functions to the correct CTATest annotation. However, this approach can be error-prone due to typos and it also hinders the ability to see all the building blocks of a test in one place. That's why the decision was made to use the approach of a container class. Since classes are allowed as annotation references, an interface was created to ensure that each container class contains all the relevant building blocks for a test case.

Having an actual reference to the method being tested ensures its existence, eliminates typos, and catches compilation errors if the method is changed or deleted. While oracle functions typically only receive the output of a test case, the different approach addresses the requirements for functions to be used as oracles in a combinatorial testing cycle based on source code annotations, as stated in RQ4: "What requirements exist for functions to be used as simple oracles and how can such oracles be used in a combinatorial testing cycle based on source code annotations?".

In the Preliminaries in Section 2, it's mentioned that CT test cases are usually executed against a separate SUT that accepts input and returns output. However, when testing functionality within a system with dependencies, return values can vary and converting everything to one type may not make sense. Working with actual classes instead of interface-based output offers advantages. Additionally, many tests in systems require setup before execution, such as preparing a specific state or executing additional commands. To provide developers with more autonomy, the decision was made to provide all the necessary building blocks for executing the test case. This allows for setup and flexibility in the return value, enhancing the test writing experience within a SUT.

These are all the annotations and building blocks needed to define a singular CT test. This leads us to the process of how we combine these building blocks to create executable tests, which will then call our oracle. This will be discussed in the next sections.

4.3 Annotation Processor

The first step in designing the whole automatic test generation process is to find a tool that fulfills the needed requirements. There are two main requirements: first, the ability to scan source code to find and extract all annotations, and second, the possibility to generate files to generate our actual test files. After a small search, the decision to use Kotlin Symbol Processing API (KSP) [Goo21] is made.

There are multiple factors why KSP is the perfect fit. First, they are promoted by the official Kotlin website and the tool is developed by Google, which reduces the chances to a minimum that the tool will not be supported or updated in the near future. It is also the successor of the widely spread and used Kotlin Annotation Processing Tool (KAPT) [Jet17], which ensures it builds on a solid foundation. Last but not least, it fulfills both requirements of scanning for annotations and creating files.

KSP uses processors as an entry point to their process. A developer can create and link an arbitrary amount of processors, and during compilation (since it is a compiler plugin), each processor will receive access to the scanned source code as well as the possibility to create additional files. In the library, the processor which handles the whole process is called CTAProcessor. A processor consists of a processor provider and the actual processor. During compilation, it first calls the provider to create the processor and afterwards calls the created processor to execute, whatever is implemented inside the process function (e.g. search for specific annotation). For a more technical in-depth look into the processor creation and implementation, see Section 5.1.

The main purpose of the processor is to find all CTATest annotations inside the code base, since these annotations are the entry point to our processing logic. The processor then validates the found annotations and passes them on to the visitors, which will be discussed in the next section. Additionally, the processor is also responsible for managing the debug mode and its accompanying logging functionality.

4.4 Visitors

To ensure a common understanding before deep diving into the different visitors, a small introduction to the visitor design pattern seems appropriate. In general, this pattern is used to separate the algorithm used to traverse and perform operations on data structures from the data structure itself. It is useful when working with complex data structures that may have different types of data that need to be processed in different ways, as it allows for the addition of new operations without modifying the underlying data structure. This pattern is especially useful when working with hierarchies of objects and is commonly used in parsers and compilers.

In a more concrete context, the code base is scanned and turned into a traversable data structure by KSP. This data structure has the typical accept function of the visitor pattern, which enables visitors to work on the data. In the case of KSP, the visitors implement an interface, which allows defining operations for every type of data in the data structure (e.g., a function, a parameter, ...). A simple pseudo code example can be seen in Listing 4.5.

Listing 4.5: Visitor Pattern Example

```
class Function {
        // fields and methods for Car
        void accept(Visitor v) {
```

```
        v.visit(this);
        }
}

class Parameter {
        // fields and methods for Car
        void accept(Visitor v) {
        v.visit(this);
        }
}

interface Visitor {
        void visit(Function f);
        void visit(Parameter p);
}

class VisitorImpl implements Visitor {
        void visit(Function f){
        print("Visiting function: " + f);
        }
        void visit(Parameter p){
        print("Visiting parameter: " + p);
        }
}
```

This example has a class representing a function and a class representing a parameter. Both classes implement an accept function, which simply calls the visit function of the visitor passed as an input argument. The visitor itself implements a visit function for each data type it supports. If a developer now calls *accept* on either a function or class, it will call the respective visit function in the visitor. This library has three different visitors, each with their own respective task to complete. As described in the previous section, the first visitor that is called is the **TestDeclarationVisitor**, which will be discussed in the next section. A small detail is that the function which searches for annotations does not return the annotation itself, but the object it resides on. In the case of the library, all **CTATest** annotations reside on a function, thus the **KSFunctionDeclaration** class, which is a class representing a function in KSP, is the entry point for the **TestDeclarationVisitor**.

4.4.1 Test Declaration Visitor

The first step in designing this visitor is to define its role and responsibilities. It is the entry point of our processing logic and as such is also the mastermind of the whole process, which delegates data to multiple sub components and ensures all the data is transformed into an executable test.

As stated above the first interaction is when the visit function for **KSFunctionDeclaration** is called. This function then triggers the following process:

1. As stated above the visitor receives the element the annotation resides on. Thus the first step is retrieve all annotations present on the element and filtering out all unnecessary annotations. This process also supports that one element contains multiple CTATest annotations.

2. Next the visitor extracts all information present in the annotation, namely the input classes and the container class.

3. All input classes are sent to the ConstraintAnnotationVisitor, which will be discussed in more detail in Section 4.4.2. This visitor extracts all information present in the input class and collects the result in an input parameter model. Important to note is that although this process suggests that multiple input classes are possible, this PoC currently only supports singular input classes. To provide proper multi input class support, some implementations which currently assume a single input class need to be adapted and additional challenges need to be overcome.

4. The resulting input parameter model is sent to the test generation tool, which generates a test set out of the IPM. More details about this process will be discussed in Section 4.6.

5. As a last step the test set is combined with additional information e.g. IPM and sent to a function which then converts the given information into executable tests. More information about this process will be provided in Section 4.6.

A more in depth implementation of this visit function can be seen in Section 5.2.

This concludes the explanation about the process of converting a single **CTATest** annotation into an executable test. Yet again this explanation is still very high level in some areas of the process and a lot is passed to additional components, which will be discussed in later sections.

4.4.2 Constraint Annotation Visitor

As described in the previous section, this visitor is called for each input class referenced in the **CTATest** annotation. The main purpose of this visitor is to parse the passed input class and create a complete input model to enable the test set generation process. One of the big differences between this visitor and the previous one is that this visitor exchanges data. As explained in the previous section, an input model class is created and passed into this visitor. This visitor also returns the input model class at the end.

The decision to create the input model class and use a different visitor implementation was made due to multiple reasons. The first reason is to create a single class that holds all definitions of an IPM for one **CTATest**. Since the class is created before starting the parsing process, this can also be easily extended to support multiple input classes, since this visitor only adds constraints and parameter definitions instead of creating a complete new IPM during each run. The biggest challenge about this transition would be to rework the way the finalized test set is translated back to the initial class. More on this process will be discussed in Section 4.6.

Another reason was the need to transfer data between different visiting functions, as well as the need for returning the parsed data back to the first visitor. Thus, the input and return value of this visitor is the input model class, which is also reflected in the function declaration, which has a return type and non-void data. All these decisions and requirements led to the final implementation of the input model class, which can be seen in Listing 4.6.

Listing 4.6: Input Model Class Implementation

```
class CTAInputModel(val fileName: CTAFileName) {
    /**
    * List of Value Constraints.
    */
    private val parameters: MutableList<CTAAbstractParameter> =
        mutableListOf()
    /**
    * List of If Constraints.
    */
    private val constraints: MutableList<CTAConstraint> =
        mutableListOf()
    /**
    * Values used to transfer data between visitors.
    */
    private var parameterType: String = ""
    /**
    * Values used to transfer data between visitors.
    */
    private var name: String = ""

    //getter and setter
}
```

The implementation is rather simple, revolving around four fields. The parameters and constraint fields each respectively hold a list of all parameter and constraint definitions. The parameterType and name field are used to transfer data between visitors and are used to enable validation between the annotation definition and the annotated parameter. A more in-depth look at this topic will be given later in this section. An interesting thing to note about the data transfer fields is that the setter always sets both fields since they are meant for temporary information transfer, and the getter cleans the field upon reading. Another interesting thing to note is that although the constraints class is a simple class containing the string constraint and nothing more, the parameter class is more complex.

Since the parameters can be quite different, as explained in Section 4.2, an interface was created which contains the common operations each parameter needs to fulfill, while the individual functionality could be expressed in the various implementations of the interface. The interface and an example implementation of one parameter can be seen in Listing 4.7.

Listing 4.7: Parameter Interface and Example Implementation

```
abstract class CTAAbstractParameter(private val name: String) {

    abstract fun getACTSString(): String

    abstract fun getVariableString(): String

    fun getParameterName(): String {
        return name
    }
}

class CTAIntParameter(private val name: String, private val
    values: List<Int>): CTAAbstractParameter(name) {
    override fun getACTSString(): String {
        return "$name(int): ${values.joinToString()}"
    }

    override fun getVariableString(): String {
        return "$name: $INT_IDENTIFIER"
    }

    companion object {
        const val INT_IDENTIFIER = "Int"
    }
}
```

From the abstract class, it can be deduced that each parameter at least has a name and implements two functions: *getACTSString* and *getVariableString*. The reason for these functions will be explained in Section 4.6. The example implementation contains additional parameters and simply calls the constructor with the mandatory name and implements the two functions. Since this parameter implementation is split on how the parameter is defined, this results in five different implementations, namely **CTABooleanPa-**

rameter, **CTAEnumParameter**, **CTAStringEnumParameter**, **CTAIntParameter**, and **CTARangeParameter**. Since the list for parameters expects any parameter that implements the interface, the range of parameters is easily extendable, since a new parameter only needs to implement the interface as well as add handling in the few places where it matters (e.g. parsing annotation to parameter).

This concludes the introduction to the input model class, its inner workings, and the reasonings behind it. Following this introduction, the next step is to explain how annotations are parsed in the aforementioned parameter and constraint definitions. As explained in the previous section, each Input Class referenced inside the **CTATest** annotation is visited with this visitor. From this follows that the *visitKSClassDeclaration* function is the first function invoked, the so-called entry point to this visitor. This function has a simple job: it searches for all annotations on the class itself, as well as all declarations (e.g. functions, properties) inside the class. It then visits all found occurrences, relaying each annotation or declaration into their respective visit function. The implementation of the *visitKSClassDeclaration* function can be seen in Listing 4.8.

Listing 4.8: ConstraintAnnotationVisitor - visit Class Declaration function

```
/**
* function implementation receiving a reference to a scanned
* class and arbitrary passed data in this case the input model
* class. It also returns the input model class to enable the
* possibility of returning the completed input model.
**/
override fun visitClassDeclaration(classDeclaration:
    KSClassDeclaration, data: CTAInputModel): CTAInputModel {
    //passing all annotation to the correct visitor function
    classDeclaration.annotations.forEach{ it.accept(this, data)
        }
    //passing all additional declarations (e.g. function,
    //variables) to the correct visitor function.
    classDeclaration.declarations.forEach { it.accept(this, data
        ) }
    return data
}
```

The annotations are visited since they could represent definitions of the input model. In general, the visit annotation function contains the main logic of parsing annotations into input model properties, since all definitions of an IPM are inside an annotation. The reason for also searching and visiting all declarations is to search for all annotations inside a class. This means the main purpose of the declaration visiting functions is to also search for all occurrences of annotations on themselves and pass them to the main annotation visit function.

In the current PoC, this visitor searches for annotations on function and property declarations (parameters). Due to the properties of the visitor design pattern, it is easy to extend the functionality in searching for more parts of a class if needed, since simply implementing a new visit function is enough. The visit function for functions is very similar to the class visit function; the only difference is that it does not search for declarations anymore, but only annotations. For properties, there is a bigger difference, since the visit function extracts the type of the property to validate if the annotation makes sense. To give a concrete example, annotating a boolean value with a **CTAInt** annotation does not make sense and will also result in errors later on when the library tries to assign an int value to a boolean property. Thus, it extracts the type via the **ResolveTypeVisitor**, which will be explained in the next section and sends the resulting type via the temporary data fields inside the input model class to the annotation visit function to provide the possibility of validating the connection between annotation and property.

This leads to the next step, which is the visit annotation function. When an annotation is visited, one of three cases can occur. The first case is that the annotation is unknown to this library, which results in ignoring the annotation. The other two cases are that the annotation is either a parameter or constraint-defining annotation. In this case, the visit function parses the annotation accordingly and adds the resulting definition to the existing input model class. After every annotation is parsed, the visitor returns the input model class back to the **TestDeclarationVisitor**. A more in-depth look into the implementation of the visit annotation function can be seen in Section 5.3.

4.4.3 Resolve Type Visitor

As explained in the previous section, this visitor has the single purpose of extracting the type of a property. Fortunately, KSP provides a simple way of extracting this information by providing a property *classKind* on every **KSClassDeclaration**. This is also the main reason for this visitor, since the **ConstraintAnnotationVisitor** already has a visit function for **KSClassDeclaration** with a different intent. Of course, it is always possible to navigate through the data structure manually, but honoring the concept of the visitor design pattern, this simple visitor was created instead.

This visitor simply extracts the *classKind* property and parses it into a usable format. The implementation can be seen in Listing 4.9.

Listing 4.9: ResolveTypeVisitor - visit Class Declaration

```
enum class ClassKind(val type: String) {
    INTERFACE("interface"),
    CLASS("class"),
    ENUM_CLASS("enum_class"),
    ENUM_ENTRY("enum_entry"),
    OBJECT("object"),
    ANNOTATION_CLASS("annotation_class")
}

override fun visitClassDeclaration(classDeclaration:
    KSClassDeclaration, data: Unit): String {
    return when(classDeclaration.classKind) {
        ClassKind.ENUM_CLASS -> "Enum"
        ClassKind.CLASS -> classDeclaration.simpleName.asString
            ()
        ClassKind.ANNOTATION_CLASS -> "Annotation"
        else -> "UNKNOWN"
    }
}
```

As seen in the example KSP differentiates between interface, class, enum class, enum entry, object and annotation. If the passed class is an enum we return Enum, if it is an annotation we return Annotation. For an actual Kotlin class we extract the class name, which also includes String or Int. For all other class kinds we return unknown, since it is a non supported class kind. From this follows that there are two possible error cases, either the type is an unsupported Kotlin type e.g. Double or the class is an unsupported class kind in general.

4.5 Test set Generator Connector

After the annotation parsing discussed in the previous step concludes, the resulting IPM is converted into a file in the format used by ACTS[YLKK13]. Details about the conversion will be discussed in Section 4.6. The reason why the IPM is needed in said format is because the CAgen[WKS$^+$20c] command line tool expects the same format for generating test sets via file input. The alternative, which would be defining the whole IPM as command line arguments, is discarded due to resulting in poor readability, comprehensibility, and a potentially extremely long command line command.

The reference to the ACTS configuration file, as well as a reference to the CAgen command line tool, serves as input for the entry point to this connector. The main purpose of this connector is to use the previously mentioned inputs, execute the library to generate a test set, and return the results for further processing. The implementation of the entry point function generateTestSet can be seen in Listing 4.10.

Listing 4.10: CTAGeneratorConnector - generate test set

```
fun generateTestSet(configPath: String, library: File): File {
    val (command, outputPath) = generateCommand(configPath,
        library)

    val response = command.runCommand()
    val outputFile = File(outputPath)
    if (isFileEmpty(outputFile)) {
        throw FileNotFoundException("Generated Testset at
            location: ${outputFile.absolutePath} not found or
            empty, Reason: $response")
    } else {
        return outputFile
    }
}
```

The first step is generating the command line command, which will be executed. Additionally, an output path is needed, defining where the resulting test set is generated. Both properties are created via the generateCommand function, which receives the configuration file path and CAgen file path and returns an executable command and the desired output path. The command created in this function has this format:

<path-to-library> -t 3 -i <acts config path> -o <output-path> –randomize

First, the absolute path to the library is inserted to tell the command line which tool to invoke. Afterwards, a strength t of the resulting test set is defined. The strength three has been chosen as a good balance between test set coverage and computing time. Due to being a PoC, which aims to automate as much as possible instead of granting a lot of customization, this strength is not configurable in this library. Next, the path to the IPM definition is set as input, and the desired output path is set as output. Lastly, the randomize flag is set to prevent the need of dealing with $don'tcare$ values. $Don'tcare$ values, depicted by a star in the test set, represent a value which is not relevant for the coverage in this exact test case, which means any value can be set. The randomize flag assigns a random value to each wildcard, which removes the need to deal with it in the library. As an example instead of resulting in *Example, 1, true, ** it will result in *Example, 1, true, "RED"*.

This generated command is then executed as a command line command in the $runCommand$ function. This function simply uses some built-in functionality from Kotlin (ProcessBuilder) to execute the command and returns the output of the tool. This output is needed to return more meaningful error messages back to the developer in case some definitions are not valid. One of the most common error cases is probably constraints, since the library relies on the developer to write valid constraints in the format of ACTS. After the execution of the command, the output file is opened and read to ensure the execution was a success. If the file does not exist or the file is empty, the library assumes an error occurred and returns the output of the tool contained in an error message. If the file exists and has content, it is returned for further processing.

4.6 File Util

The main responsibility of the **File Util** is to handle all aspects of file creation, as implied by its name. The main responsibilities are creating the ACTS format input model configuration, retrieving the test set generation tool, and transforming all inputs collected from previous steps into executable tests and generating them. The first two actions are performed before running the command line tool discussed earlier, while the last action is the final step in converting annotations to executable tests.

The first action, creating the ACTS format input model, involves a two-step process. First, a new file is generated and then filled with the translated configuration from the received input model. The implementation and an example in ACTS format can be found in Listing 4.11.

Listing 4.11: File Util - ACTS example and generate ACTS file

```
[System]
Name: TestClassCTTest

[Parameter]
boolean(boolean): true, false
enum(enum): Example1, Example2
numberWithArrayOfValues(int): 0, 1, 2, 3, 4, 5
numberWithRange(int): 0, 1, 2, 3, 4, 5
stringEnum(enum): RED, GREEN, YELLOW

[Constraint]
enum = "Example1" => stringEnum != "RED"
numberWithRange = 2 => numberWithArrayOfValues = 4
```

```
fun generateACTSFile(inputModel: CTAInputModel): String {
    val filename = inputModel.fileName.
       getConfigFileNameNoExtension()
    val file = codeGenerator.createNewFile(Dependencies(false),
       "", filename, "txt")

    val actsTemplate = createACTSTemplate(inputModel)
    file.appendText(actsTemplate)
    file.close()

    val configFile = codeGenerator.generatedFile.find { it.name
       == inputModel.fileName.getConfigFileNameWithExtension() }
```

```
            ?: throw FileNotFoundException("Generated Config
                File not found")
    return configFile.absolutePath
}
```

As seen in the example, the configuration file is very straightforward. The system name is simply the name from the input model and has no real effect on the creation of the test set. Next are the parameter definitions, one parameter each line written in the format *<name>(<type>): <valid values>.*. At last, the constraints are also inserted one constraint per line. This should also clear up the need for the *getACTSString* function described in a previous section, as it is a simple helper function that automatically translates an extracted parameter into the expected format.

The function itself first creates a new file with the built-in code generator. Afterwards, the input model is translated into the needed format and written into the newly created file. This translation is done via a string template, which is a built-in functionality in Kotlin. Templating will be discussed later on when it is used more extensively during the translation to executable tests. In this case, the template simply has the basic structure of the configuration file and inserts parameters and constraints in the respective areas, one line at a time.

The second action, which is retrieving the test set generation tool, is a bit special. The need for this function arises from the need to have the test set generation tool present during execution. The simplest way would be to expect the developer to have the tool downloaded and made available in the command line (e.g., via path variable). This library has a different approach of bundling the tool as a resource and using this resource for generation. The problem with this approach is that the tool is bundled inside the JAR during the execution inside another system. Thus, the **File Util** creates a file and copies the tool inside this file. On consecutive access, it returns the finalized file. The advantage of this approach is that the developer doesn't need to have the command line tool prepared, while the library can be completely certain that the tool exists in a known location.

The third action, which is transforming all inputs collected from previous steps into executable tests, is the most complex of the three actions. It essentially ties all the information from the previous steps together to create usable test cases. The information it ties together consists of the list of referenced input classes, the referenced container class, the generated test set, and a name for the test class. The main challenges this method has to solve are how to translate the generated test set into usable input data and how to generate executable tests, which also ties into the research question 2, which revolves around how combinatorial test suites can be generated from existing Kotlin sources with annotations and executed in the context of continuous integration. This topic will also be answered in the following paragraphs.

The function itself is very similar to the generation of the ACTS format configuration file. It determines a suitable package and file name for the test file based on the referenced input classes. Then, a new file is created via the code generator. Afterwards, the test class content is realized via a template, written into the generated file, and saved. From this explanation, it can be easily seen that the whole logic to overcome the above-mentioned challenges lies inside the function that realizes the template. But before designing the previously mentioned template, additional decisions had to be made. How would the template result in executable tests that can be executed in the context of continuous integration?

The basis of the decision was the need for KSP for Gradle [Dav19]. Gradle is a powerful and widespread build tool in the Kotlin and Java world, which offers a lot of functionality for dependencies, building, publishing, and much more. A typical Gradle build involves compiling the codebase and, among other tasks, also running referenced test cases inside the test package. Besides the build task, Gradle also offers the option to selectively run the test task, which executes all tests without running a complete build, provided that the codebase has been compiled before. Additionally, KSP also offers its own task via Gradle, which executes the compilation task alongside any defined processors. This results in solid support for continuous integration, as developers can decide to either simply run a full build, which would compile, invoke KSP, and run generated tests, or run specific steps as needed, e.g., for an integration test scenario.

This decision leads to the question of how to generate tests that are executable by Gradle and also recognized as part of the test package. The answer to the first question is using JUnit 5 tests [GS17]. JUnit is one of the most widespread testing frameworks found in nearly every Kotlin or Java project, which is a big plus since most developers are familiar with the framework and no new technology has to be introduced or explained. Regarding the second question, the solution is to add the folder that houses the generated test files to the test package to ensure they are found and executed by Gradle. Alternatively, the generated tests can also be executed manually. This concludes the answer to the research question of how to generate CT test suites in the context of continuous integration 2.

The above decisions result in the need to create a test template that uses JUnit 5. Since this template is a lot more complex than the configuration file, the decision was made to create a TestTemplateSource, which is a class that holds all the needed information to fill the template. This class has the advantage that all the information needed is inside a single object, and the possibility to create utility functions enhances the readability of the template parsing part, as singular functions are called instead of mutating the data as needed inside the template. This process is realized in the createTestTemplate function, which creates the source and parses the template. The implementation can be seen in Listing 4.12.

Listing 4.12: File Util - generate test file

```
fun createTestTemplate(
    testSet: CTATestset,
    testName: CTAFileName,
    containerClass: CTAFileName,
    classesToTest: List<KSClassDeclaration>,
    part: Int
): String {
    val imports = TestTemplateImports(classesToTest, testSet.
        parameters)
    val source =
        TestTemplateSource(testSet, imports, testName,
            containerClass, classesToTest.map { it.simpleName.
            asString() }, part)

    return parseTemplate(source)
}
```

The first step of the process is to identify all additional imports this resulting test class has to import. This involves imports of all input classes, the container class, as well as all referenced enums. Next, the imports and all additional available information are passed into the source class. Besides holding and offering all the inserted information, this class also has two important functions. To understand the need for these functions, a bit more explanation of the test template has to be done. The overall idea of the test template was to use parameterized tests, which is a functionality offered by JUnit. Parameterized tests have a source, which can be various options (e.g. function, CSV file). Each entry of the source is executed as a test case. What makes these tests special is that they all share the same implementation, with only the input changing. This is ideal since the sole purpose of the generated test is to create the input data for a test case and send it to the oracle and await the decision of the oracle if the test passed or not.

This leads to two possible approaches. One way would be that the parameterized test receives the CSV test set as source, which means the function starts with a number of values and the template needs to provide a function that can then translate all the values to a usable class object. The other way would be to translate the test set during the processing, which results in a list of class objects as input, enabling the possibility of simply calling the oracle with the passed input. This PoC decided on the second possibility since it removes the need for the translating function and improves the speed, as the test is confronted with already instantiated objects. Either approach has some size limitation because at one point too many inputs will result in the function becoming too big (a single method can consist of at most 65536 bytes of bytecodes), which is an active limitation of Kotlin.

This leads back to the two functions mentioned above. Both functions are used to create a helper function, which is used to instantiate a class object. This helper function is then used to create all the test cases for the test. A simplified representation of the template can be seen in listing 4.13.

Listing 4.13: File Util - test template

```
<static imports e.g. junit, basic kotlin functions>
<dynamic imports e.g. input class, container class>

 class <class name> {

    @ParameterizedTest
    @MethodSource(<test case source>)
    fun testing<class name>(<inputs>) {
        <invoke oracle>
        <if oracle negative response -> fail test>
    }

    companion object {
        <test case source> = Stream.of(
            <invoking helper function with inputs of one test
                case>
            <invoking helper function with inputs of one test
                case>
            ...
        )

   <helper function which creates class with passed inputs>
    }
}
```

As seen in the template approach, the helper function is created and each test case is then converted into a class object via the helper function. The stream of these class objects are then used as input for the parameterized test, where they are passed to the oracle function. The resulting template are executable JUnit 5 tests. This concludes the process of how annotations are used to create executable tests. The next section contains a list of limitations of this library, since this work is still a PoC and is not fully fleshed out.

4.7 Limitations

There are a multitude of reasons for limitations in this work. Some limitations arise due to architectural decisions or limitations of used tools. Other limitations are due to being a PoC, which means some limitations are set knowingly. The main limitations of this work are:

- **Kotlin** Using Kotlin brings some limitations, such as the requirement for projects to use Kotlin and Gradle, along with other additional restrictions.

  - **Version** - The Java or Kotlin version is a big topic in every system. Thus the targeted SUT needs to at least have the same Kotlin version as the library. This is an active limitation of the language/compiler itself due to different functionalities offered between versions, which could potentially not be supported in older versions.
  - **Annotations** - Kotlin has great support for annotations, allowing them to be used on classes, functions, and variables. However, these annotations do have significant limitations, which have led to unconventional approaches in the defined format of this work.

- **Oracle Problem** - This is a commonly known limitation in CT. Most of the time a proper knowledge of the inner workings of the system is needed to create a proper oracle. This will also come up later on in Section 6, but inferring how a test case should behave solely by the input data is often not possible. This fact results in difficulties for testers to write meaningful oracles without the help of a developer.

- **State in OOP** - Object-oriented programming (OOP) is a programming paradigm that organizes code around objects with data and behaviors, promoting code organization and reusability. Kotlin's focus on OOP leads to the limitation of needing state (of previously mentioned objects) in OOP systems, affecting the PoC. In contrast to typical approaches where input is simply sent against an interface and each test case is meant to be stateless, tests against function often need some kind of state setup to make sense. This state is oftentimes also part of a test case, because depending on the previous state there could be multiple outcomes, thus it would make sense to also take these possible states into account during test set generation. Currently the only possibility is to integrate the state variables into the input class, which is used for generating the test and then extracting the values from the input class oneself for setup. This clashes with the idea of a developer creating input classes for functions and annotating them without altering the initial class. In the future there could be a way to define or generate a state as well.

- **Instantiation of a class** - At one point or another the generated test has to instantiate the class with the given test case values to enable the use of it in a test. In the context of the PoC the class needs to have an empty constructor and variables have to be reassignable. Other possibilities could have been to either parse the constructor and decide depending on the structure of the constructor, or break open variables via reflection to add the values. This would have inflated the scope and since the instantiation is not the main focus of this work the above limitation was accepted.

- **IPM limitations** - In the current PoC the input classes are limited to one class, which could be easily adapted in future works and was an active decision to limit the scope of the work. An additional limitation discovered during the evaluation in section 6 is that there is currently no possibility to create multiple instances of the same input class. As an example, if the system has a class representing a point in a coordinate system and a developer wants to create a test case involving 3 points there is no easy way to do that without creating a class and duplicating the variables. Although this is also not supported by the test generation tool, it could be provided and supported by a KSP processor.

- **CAgen ACTS limitation** - As explained in a previous section CAgen[WKS+20c] expects the same format as ACTS[YLKK13], but it does not support the full range of operations. The upside of CAgen is speed, which is very important to not prolong the compilation process by a lot. If there is ever the need to support the full range of operations, the change to ACTS could be easily done by changing the tool and command and simply using the same configuration file. But in the context of this PoC this is an active limitation, by the decision to use CAgen as a test set generation tool.

CHAPTER 5

Implementation

This chapter highlights and explains in depth technical details and processes revolving around the whole library workflow.

5.1 CTAProcessor

This section contains technical details about CTAProcessor creation and implementation.

5.1.1 CTAProcessor Creation

To create and link a KSP processor the first step is to create a processor, which implements an interface from KSP called SymbolProcessor. This interface contains a single method called process, which is then called during the compile process. This method receives a Resolver as an argument, which is a class that allows to search and access all the annotations inside the code base. Additionally the code generator which is also provided by KSP is used as an additional argument for the generator. A basic example of the processor without functionality can be seen in Listing 5.1.

Listing 5.1: Basic Processor KSP

```
class CTAProcessor(
    private val codeGenerator: CodeGenerator
) : SymbolProcessor {

    /**
    * @param resolver class from ksp which can be used to search
        for annotations
    * @return list of invalid annotations which could be used in
        further processors
    */
    override fun process(resolver: Resolver): List<KSAnnotated>
        {
            //functionality
    }
}
```

The next step in the processor creation process is to create a provider. Similar to the processor the provider also implements a provided interface called SymbolProcessorProvider, which contains a single method create. The create function has a SymbolProcessorEnvironment as a single input parameter and expects a SymbolProcessor as return value. The passed environment provides access to classes and information relevant for the processing process, such as the Kotlin version, compiler version or a code generator. To implement the create function we used the code generator to create the previously created CTAProcessor and pass this processor back as return value. The final provider can be seen in Listing 5.2.

Listing 5.2: Processor Provider KSP

```
class CTAProcessorProvider : SymbolProcessorProvider {
    override fun create(
         environment: SymbolProcessorEnvironment
    ): SymbolProcessor {
         return CTAProcessor(environment.codeGenerator)
    }
}
```

The last step to create a processor is to link the provider to KSP, which enables KSP to create and call the processor. To link the provider to KSP a reference to the provider in a specific resource file is needed. The specific file is located under *resources/META-INF/services/com.google.devtools.ksp.processing.SymbolProcessorProvider* and simply adding package and class name (e.g. tuwien.cta.processor.CTAProcessorProvider) is enough. This concludes the creation and linking process, from now on KSP calls the provider during every compilation process.

5.1.2 CTAProcessor Implementation

Next is the implementation of functionality. The actual processing of annotations is done via visitors which will be explained in depth in the next section. Thus the main purpose of the processor is to find specific annotations, which are CTATest annotations in the context of this library, and validate and pass them to relevant visitors. Additionally it also searches for the debug annotation and creates a relevant **Logging Util**, which is a utility class designed to help with logging relevant information, if set. It ensures to only log if the **CTADebug** annotation is present and also creates the needed logfile. The finalized process function can be seen in Listing 5.3 and works the following way:.

Listing 5.3: Processor process function KSP

```
1  override fun process(resolver: Resolver): List<KSAnnotated> {
2          //check if debug mode is enabled
3          val isDebug = resolver.getSymbolsWithAnnotation("tuwien.
               cta.annotation.utility.CTADebug").count() > 0
4
5          //get all CTATest annotations
6          val symbols = resolver.getSymbolsWithAnnotation("tuwien.
               cta.annotation.test.CTATest")
7
8          //process all valid CTATest annotations
9          if (symbols.count() > 0) {
10             val loggingUtil = createLoggingUtil(isDebug)
11
12             symbols
13                 .filter { it is KSFunctionDeclaration && it.
                       validate() }
14                 .forEach { it.accept(TestDeclarationVisitor(
                       codeGenerator, resolver, loggingUtil), Unit)
                       }
15
16             loggingUtil.close()
17         }
18
19         //return all invalid symbols for possible future
               processing
20         return symbols.filter { !it.validate() }.toList()
21 }
```

- **Line 3, 10, 16:** Checks if any CTADebug annotations are present with the use of the *getSymbolsWithAnnotation* function. Depending if an annotation is present either a dummy or an actual **Logging Util** is created. Finally after the whole annotation processing the **Logging Util** is cleaned up.

- **Line 6, 9:** Checks for any CTATest annotations via the *getSymbolsWithAnnotation* function. If there are any the annotation process is started otherwise it does nothing. This also correlates to using CTATest as enable annotation mentioned in Section 4.2.

- **Line 12,13,14,20:** First it filters out invalid annotations or annotations placed in unexpected locations, since the CTATest annotation should only be placed on functions, then it passes each valid annotation to the TestDeclarationVisitor which is described in more depth in Section4.4.1. At last it returns invalid annotations to provide the possibility for other processors to process it.

The logging util itself is a simple wrapper around a logging file, which provides logging functions which only log, when a file exists. The way the *createLoggingUtil* function works is it only creates a logging file if the isDebug boolean is true and thus the logging util only logs if a CTADebug annotation is present.

5.2 TestDeclarationVisitor

The implementation of the visit function can be seen in Listing 5.4 and works the following way:

Listing 5.4: TestDeclarationVisitor - visit KSFunctionDeclaration Implementation

```
 1 private val knownAnnotations = listOf("CTATest")
 2
 3 override fun visitFunctionDeclaration(function:
       KSFunctionDeclaration, data: Unit) {
 4     val functionName = function.simpleName.asString()
 5     if (visited.contains(functionName)) {
 6         return
 7     }
 8     visited.add(functionName)
 9
10     function.annotations
11         .filter { knownAnnotations.contains(it.shortName.
              asString()) }
12         .forEach { it.accept(this, data) }
13 }
```

- **Line 4-8:** The function name gets extracted and checked against a static list which contains all previously visited functions. If the function name is contained inside the list we stop the process since we do not want to process the same function multiple times. Otherwise we add the function to the list and continue.
- **Line 10-12:** We extract the list of all annotations present on this function and filter it to only contain relevant annotation, which are only CTATest annotations and then accept the remaining annotations with the same visitor, which results in execution of the visit function for annotations for each CTATest annotation.

This leads us to the visit function for KSAnnotation, which is the function which contains the logic of processing a CTATest function from start to finish. The implementation of the annotation function can be seen in Listing 5.5 and works the following way:

Listing 5.5: TestDeclarationVisitor - visit KSAnnotation Implementation

```
override fun visitAnnotation(annotation: KSAnnotation, data: Unit){

    val containerClass = extractContainerClass(annotation) ?: return

    val inputModelClasses = extractInputModelClasses(annotation)

    if (inputModelClasses.isEmpty()) {
        return
    }

    // use one of the specified classes for testing for name and package
    val testName = extractClassNameAndPackage(inputModelClasses[0])

    var inputModel = CTAInputModel(testName)

    for (inputModelClass in inputModelClasses) {
        inputModel = inputModelClass.accept(
            ConstraintAnnotationVisitor(codeGenerator, resolver, loggingUtil),
            inputModel
        )
    }

    val pathToACTSFile = fileUtil.generateACTSFile(inputModel)
    val libraryFile = fileUtil.getLibraryFile()

    val testSetFile = generatorConnector.generateTestSet(pathToACTSFile, libraryFile)

    val testSetEntries = csvReader().readAll(testSetFile)
    val testSet = CTATestset(inputModel.getParameters(), testSetEntries)

    fileUtil.generateTestFile(testSet, testName, containerClass, inputModelClasses)
}
```

- **Line 3:** Extracting the container class with a helper function, which will be discussed in more detail in the next paragraphs. If no valid container class is found the function returns, since without a container class we have neither a SUT nor an oracle function.
- **Line 5-9:** Extracting the input classes with another helper function, which will also be discussed in the next paragraphs. If there exists no input classes the function returns, since without an input class no input model can be created.
- **Line 12-14:** Extracting package and class name to use for the auto generated tests as well as creating an input model class which is a class which represents an input model and is used during the extraction of parameter definitions and constraints.
- **Line 16-21:** Iterating over each extracted input class, visiting each entry with the ConstraintAnnotationVisitor, which processes all parameter and constraint definition annotations and returns an input model.
- **Line 23-24:** The previous step extracted the complete input model which is now ready to be used as input for the test set generation tool. The tool needs a configuration and the library also needs to know where the tool exists, to call it, which is done in these two lines.
- **Line 26:** After gathering enough information to execute the tool, this line executes the tool and receives the generated test set as a file reference.
- **Line 28-29:** The retrieved test set file is a CSV file, which is parsed with a CSV reader library. For this purpose a library named *kotlin-csv* [doy21] was used. After extracting all lines of the CSV file the result is saved in the CTATestset.
- **Line 31:** As the last step a function which generates the actual test file is called with a combination of the generated test set, input model, container class and a test name.

The implementation of the container class can be seen in Listing 5.6 and works the following way:

Listing 5.6: TestDeclarationVisitor - extract Container Class implementation

```
private fun extractContainerClass(annotation: KSAnnotation):
    CTAFileName? {
    val container = annotation.arguments.find {
        val argumentName = it.name
        argumentName != null && argumentName.asString() ==
            CTA_CONTAINER_ARGUMENT
    } ?: return null

    val containerType = container.value as KSType
```

```
 8      val containerDeclaration = containerType.declaration
 9      return if (containerDeclaration is KSClassDeclaration) {
10          extractClassNameAndPackage(containerDeclaration)
11      } else {
12          null
13      }
14  }
```

- **Line 2-5:** Searching the annotation for the container argument and extracting it, if there is no argument present the function returns null
- **Line 7-13:** The declaration of the argument is extracted and a check is performed if the container argument is indeed a valid class declaration. If the declaration is valid the function returns class and package name of the container class otherwise it returns null

The main difference for input classes is that the function searches for a different annotation argument as well as that input classes are a vararg, which means they are retrieved in a list. This means the function handles a list instead of a single specific element. Otherwise the implementation is the same.

5.3 ConstraintAnnotationVisitor

The implementation of the visit annotation function can be seen in Listing 5.7.

Listing 5.7: ConstraintAnnotationVisitor - visit Annotation function

```
override fun visitAnnotation(annotation: KSAnnotation, data:
    CTAInputModel): CTAInputModel {
    return if (annotation.shortName.asString() ==
        CONSTRAINTS_ANNOTATION_NAME) {
          parseConstraints(annotation, data)
    } else if(KNOWN_PROPERTY_ANNOTATIONS.contains(annotation.
        shortName.asString())) {
          parseValueConstraint(annotation, data)
    } else {
        data
    }
}
```

To provide better scoping and readability the parsing functionality is split into 2 separate functions. The parseConstraints function is simpler, since there are less implications. The implementation can be seen in Listing 5.8. First it searches for the if constraint argument, which is the list of CTAIfConstraint inside the CTAConstraints annotation as described in Section 4.2. It then iterates through each if constraint, extracting the constraint inside, checking it for unsupported operations (more about this in Section 4.5) and if no errors occur adding it to the input model class.

Listing 5.8: ConstraintAnnotationVisitor - parse Constraint

```
private fun parseConstraints(annotation: KSAnnotation, data:
    CTAInputModel): CTAInputModel {
    val ifConstraintArgument = annotation.arguments.find { it.
        name?.asString() == IF_CONSTRAINTS }
    if (ifConstraintArgument != null) {
        val valuesList = (ifConstraintArgument.value as List<*>)
            .filterIsInstance<KSAnnotation>()
        valuesList.forEach { ifConstraint ->
            val constraint = ifConstraint.arguments.find { it.
                name?.asString() == IF_CONSTRAINTS_VALUE }
            if (constraint != null) {
                val value = constraint.value as String
                checkForUnsupportedOperations(value)
                data.addConstraint(CTAConstraint(value))
            }
        }
    }
    return data
}
```

The property parsing is a bit more complex, since there is validation of the connection between annotation and property type as well as the diversity of parameter definition annotations, resulting in the need to write a parser for each specific annotation type. The implementation can be seen in Listing 5.9. After extracting the type and name received from the previous visit function, the first step is to verify if the type is known in this library. Currently the only valid properties to annotate are Int, Enum, Boolean and String. How this type extraction process works and the need for UNKNOWN is described in the next section.

Listing 5.9: ConstraintAnnotationVisitor - parse Property

```
enum class CTAType {
    Int,
    Enum,
    Boolean,
    String,
    UNKNOWN
}
```

```
private fun parseValueConstraint(annotation: KSAnnotation, data
    : CTAInputModel): CTAInputModel {
    val (typePayload, namePayload) = data.getPayload()

    val type: CTAType
    try {
        type = CTAType.valueOf(typePayload)
    } catch (e: Exception) {
        return data
    }

    val validated = annotation.validateAnnotatedType(type)
    return if (validated) {
        val propertyConstraint = annotation.generateParameter(
            type, namePayload)
        data.addParameter(propertyConstraint)
        data
    } else {
        data
    }
}
```

If the extracted type is a valid type the next step is to validate if the correct annotation is set on the correct property via the validateAnnotatedType function. This function simply checks which type is passed e.g. Enum and verifies if the set annotation is the correct one e.g. CTAEnum. After this validation is passed the next step is to parse the annotation to the respective parameter class. This is done via the generateParameter function, which checks for the type of the annotation and then calls the specific parsing function for the respective annotation type. The function implementation as well as the implementation of one specific parsing function can be seen in Listing 5.10.

Listing 5.10: ConstraintAnnotationVisitor - generate Parameter

```
fun KSAnnotation.generateParameter(type:CTAType, name: String):
     CTAAbstractParameter {
    return when(type) {
        CTAType.Int -> ctaParameterGenerator.generateIntParam(
            this, name)
        CTAType.Boolean -> ctaParameterGenerator.
            generateBooleanParam(name)
        CTAType.Enum -> ctaParameterGenerator.generateEnumParam(
            this, name)
        CTAType.String -> ctaParameterGenerator.
            generateStringEnum(this, name)
        CTAType.UNKNOWN -> throw InvalidDataTypeException("
            Cannot generate Parameter from type Unknown")
    }
}

fun generateStringEnum(annotation: KSAnnotation, name: String):
     CTAAbstractParameter {
    val valuesArgument = annotation.arguments.find { it.name?.
        asString() == ENUM_VALUES }
    if (valuesArgument != null) {
        val valuesList = (valuesArgument.value as List<*>).
            filterIsInstance<String>()
        if (valuesList.isNotEmpty()) {
            return CTAStringEnumParameter(name, valuesList)
        }
    }
    throw InvalidAnnotationException("missing argument on
        annotation")
}
```

While the generateParameter function is a simple when block which checks for the passed type and calls the relevant parsing function, the parsing function itself is pretty similar to the parsing function in Listing 5.8, searching for expected annotation arguments and constructing the respective class with the extracted information. The resulting class is then added to the input model class as parameter definition and the parsing process is complete.

CHAPTER 6 Evaluation

The chosen approach to evaluate the finalized PoC was to integrate it into an existing code base and write CT test cases for it. This evaluation should ensure that the library can fulfill the basic use case of defining an IPM via annotations and generating CT tests. The first step in this evaluation process was to search for a suitable project. As described in Section 4.7, the project had to be a Kotlin project with a matching Kotlin version and Gradle. It should be a medium sized codebase project to provide enough content to write multiple meaningful test cases. After attempting to integrate some projects as a test it also became apparent that a requirement was that the code base worked with self defined data structures, since if the code base relied on third party data structures, there would be no possibility to annotate these data structures. A popular example in the Kotlin context would be Android apps, since they often work with a variety of predefined classes and data (e.g. Activities and Intents), which limits the possibility to annotate and define meaningful CT tests.

The search resulted in the project KotCity [kot18] chosen as evaluation target. As explained in their README KotCity is a city simulator, which aims to emulate the excitement and possibilities of previous popular city simulators such as SimCity. The state of development was still on alpha level during evaluation, which was an upside since a full blown city simulation would probably be far from a medium size project. Another upside of this simulation was that it was written from scratch, resulting in an optimal testing field for the PoC.

The next step in the evaluation was to integrate the PoC into KotCity. Following the written guide in the README provided in the PoC the integration posed no big hurdles and went without any major issues. Before starting to annotate and generate CT tests it was a must to familiarize oneself with the code base and identify potential points which were deemed worth testing. Luckily the authors of the simulation already wrote some generic unit tests, which were used as inspiration to choose targets for the evaluation. The final decision was made to focus on three tests related to different aspects of the city simulation,namely a build collision test, a pathfinding test, and an upgrading test. An overview of the three evaluation tests containing parameter and constraint count as well as test set size can be seen in Table 6.1. Each evaluation test will be explained in depth in the following paragraphs.

Testing Evaluation Result			
Test Name	Parameters	Constraints	Testset Size
Collision	3	0	5766
Pathfinding	6	6	2945
Upgrader	12	3	6128

Table 6.1: Evaluation Test Results

The first test revolved around the collision of buildings. The simulation always starts with a clean state, which means the starting point is an empty field of varying size containing blocks. Each block has a repertoire of commands it can execute and one of these commands is the build command, which builds a structure of a given type at the given location. Structures vary in a multitude of properties, one being the size of the building, which could be only 1 block or multiple. This can lead to collision of buildings when either attempting to build 2 buildings in the same location or building a larger building too close to another building. In these cases the latter building should not be built. This was the content of the first test.

It had 3 parameters, namely the x and y coordinates of where to build and the type of structure which should be built. Unfortunately there was no class containing all needed values and for the type there was no class at all, since in all test cases using the type just wrote a plain text type referencing a JSON file. Thus a helper class was created to define the IPM and the values were reassigned in the oracle. After annotating the test with the needed input class and container class, the test set was generated containing 5766 test cases. The annotated class as well as the generated ACTS format configuration and a sample of the test set can be seen in Listings 6.1,and 6.2. For each coordinate, the test constructed a building at a given location and then attempted to build a second one on the exact same spot or too close by. It was confirmed that none of the subsequent constructions were successful. Beside the needed workaround for the IPM due to missing definition of properties there were no other issues and all test cases passed without an issue.

Listing 6.1: Evaluation - Collision Test - Input Class

```
class CollisionInputClass {

    @CTAEnum(values = ["slum1", "slum2", "cheap_house", "2796
        _charla_lane", "1278_sherman_street", "277_irving_place
        "])
    var buildingType: String = "placeholder"

    @CTAInt(from = 0, to = 30)
    var x: Int = 0

    @CTAInt(from = 0, to = 30)
    var y: Int = 0
}
```

Listing 6.2: Evaluation - Collision Test - ACTS format config and sample

```
[System]
Name: CollisionCTACTTest

[Parameter]
buildingType(enum): slum1, slum2, cheap_house, 2796_charla_lane
    , 1278_sherman_street, 277_irving_place
x(int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30
y(int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30

[Constraint]

___________________________________________________________________

Sample Test Cases:
slum1,0,0
slum2,0,0
cheap_house,0,0
```

The second test revolved around pathfinding in particular finding a way out of the city. From the referencing test it could be inferred that pathfinding only worked if there was a road leading out of the city. This resulted in the test case of building a road and then given a point testing if the pathfinding finds a way out. To define the IPM a helper class was needed again, since the need for multiple instances of the same type arose. In the case of the test case multiple points in the city were needed which consist of x and y coordinates. In total the IPM consisted of six parameters: two parameters describing x and y coordinates of starting point for pathfinding and four parameters describing x and y coordinates of starting and end point of the road which is built. Additionally it consisted of six constraints, which ensured that the generated road was a valid road, meaning it spanned from one side of the city to another. The resulting test set had 2945 test cases. The resulting input class, ACTS format configuration and a sample of the test set can be seen in Listings 6.3, 6.4.

Listing 6.3: Evaluation - Pathfinding Test - Input Class

```
@CTAConstraints(
    CTAIfConstraint("xRoadStart $CTEQUALS xRoadEnd $CTIMPLIES
        yRoadStart $CTNOTEQUALS yRoadEnd"),
    CTAIfConstraint("yRoadStart $CTEQUALS yRoadEnd $CTIMPLIES
        xRoadStart $CTNOTEQUALS xRoadEnd"),
    CTAIfConstraint("xRoadStart $CTEQUALS 0 $CTIMPLIES xRoadEnd
        $CTNOTEQUALS 0"),
    CTAIfConstraint("yRoadStart $CTEQUALS 0 $CTIMPLIES yRoadEnd
        $CTNOTEQUALS 0"),
    CTAIfConstraint("xRoadStart $CTEQUALS 30 $CTIMPLIES xRoadEnd
        $CTNOTEQUALS 30"),
    CTAIfConstraint("yRoadStart $CTEQUALS 30 $CTIMPLIES yRoadEnd
        $CTNOTEQUALS 30"),
)
class PathfindingInputClass {

    @CTAInt(from = 0, to = 30)
    var xStart: Int = 0
    @CTAInt(from = 0, to = 30)
    var yStart: Int = 0

    @CTAInt(values = [0, 15, 30])
    var xRoadStart: Int = 0
    @CTAInt(values = [0, 15, 30])
    var yRoadStart: Int = 0

    @CTAInt(values = [0, 15, 30])
    var xRoadEnd: Int = 0
    @CTAInt(values = [0, 15, 30])
    var yRoadEnd: Int = 0
}
```

Listing 6.4: Evaluation - Pathfinding Test - ACTS format config and sample

```
[System]
Name: PathfindingCTTest

[Parameter]
xRoadEnd(int): 0, 15, 30
xRoadStart(int): 0, 15, 30
xStart(int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
    15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
    30
```

```
yRoadEnd(int): 0, 15, 30
yRoadStart(int): 0, 15, 30
yStart(int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
    15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
    30

[Constraint]
xRoadStart = xRoadEnd => yRoadStart != yRoadEnd
yRoadStart = yRoadEnd => xRoadStart != xRoadEnd
xRoadStart = 0 => xRoadEnd != 0
yRoadStart = 0 => yRoadEnd != 0
xRoadStart = 30 => xRoadEnd != 30
yRoadStart = 30 => yRoadEnd != 30
```

```
Sample Test Cases:
0,15,0,0,15,0
15,30,0,15,30,0
30,0,0,30,0,0
```

Besides the workaround for the IPM, the most apparent limitation which came up during this test was the oracle problem. This issue is discussed more in depth in Section 4.7, but the gist is that often in depth knowledge of the system is needed to write a meaningful oracle. Simply expecting pathfinding to find a path when a road exists, led to a lot of failing tests. A closer look into the pathfinding functionality was needed to write a proper oracle which could decide if the current scenario should find a path or not. It turned out that if no road was nearby, meaning in a 3 block range, the pathfinder just gave up. Additionally it could not handle diagonal roads; only horizontal or vertical roads were valid. Taking this into account led to all tests passing.

The third and last test was about upgrading structures. Each tick of the simulation the buildings had the chance to be upgraded if the building had enough goodwill. This test had a lot of parameters, because of the need of a state (more about this topic in Section 4.7). As explained in the first test the city always starts with a clean state, but many test cases need some kind of setup to make sense. An upgrader upgrading nothing because the city is empty is not a meaningful test case and while a static setup can be done easily it would waste the potential of CT to not also reference the setup state in the input model to cover multiple different starting scenarios. This was also apparent in the previous test case where the test needed a road, which strictly speaking was also part of a setup and not of the pathfinding test per se. One way or another a future work could be to enable the possibility to craft varying states for test cases separated from the oracle function. In this specific case the parameters represented three buildings each with their own x and y coordinate, build type and goodwill. Additionally the IPM had 3 constraints which ensured that the three buildings would never be in the same location. The IPM resulted in 6128 test cases. In addition to the state issue this test also suffered from the oracle problem, since a deeper inspection was needed to find out multiple structures do not start with level one but higher as well as the upgrader only upgrading one building although the docs stated it could upgrade up to three buildings. Taking this into account led to all test cases passing.

This concludes the evaluation of the PoC. Apart from discovering some known and unknown limitations, which resulted in workarounds, there was overall no case which could not be modeled or tested and the test generation and execution went without any issues. The flow of defining the input model with an empty oracle, generating the test cases and then implementing the oracle and tweaking the IPM felt like a very natural process.

CHAPTER 7

Conclusion

In conclusion, while combinatorial testing provides upsides such as guarantees about input space coverage, it still struggles to make an emergence in the mainstream world of software development. This work tried to address one of the biggest downsides in current CT approaches, namely the disconnect between the input parameter model and the actual targeted software system. The disconnect results in an unnatural change process for the IPM, which often results in an IPM depicting a different state than the actual system.

To counteract this issue this work introduced methods to define IPMs and CT tests via annotations, declare methods under test and define and provide a simple oracle functions, which ensures IPM and target system are connected. Additionally, a small literature study was conducted investigating papers tackling similar issues and comparing their methods to methods in this work. We presented the implementation, challenges and limitations of the defined methods and evaluated the resulting PoC against an external target system, which demonstrated that the PoC is capable of annotating meaningful IPMs, generating CT test suites and defining simple oracles for each test.

This work shows an approach of combining annotations and source code to create IPMs. It provides an entry point into the topic of creating a connection between IPMs and targeted systems and enables developers to create CT tests instead of a small count of experts. The empowerment of developers, as well as the easier setup and change process could be important steps to push CT into mainstream software development.

There is a lot of potential for future work in my opinion. The most natural idea would be to refine the PoC. An example of refinement would be the removal of easier limitations, such as multi class support or testing of web interfaces. Another potential option would be to use the same approach with another programming language and compare the performance, upsides and downsides of both approaches. One of the biggest downsides was the limitation due to programming language, which naturally leads to finding a language independent solution as possible future work. It remains questionable if this can be achieved without sacrificing the current level of automation. There could also be an investigation if there is another way to enable developers to define IPMs inside their own system, which has more upsides or fewer downsides than using annotations. Lastly, the limitation regarding state in OOP systems could be addressed, eliminating the need for misusing the oracle for setup and cleaning tasks.

List of Figures

List of Tables

Listings

Acronyms

PoC	Proof of Concept.
SDLC	Software Development Lifecycle.
CT	Combinatorial Testing.
IPM	Input Parameter Model.
CI/CD	Continuous Integration/Continuous Deployment.
SuT	System under Test.
CST	Combinatorial Security Testing.
CA	Covering Array.
XSS	Cross-site-scripting.
IPO	In-Parameter-Order Algorithm.
CASA	Covering Arrays by Simulated Annealing.
GUI	Graphical User Interface.
MBT	Model-Based Testing.
QA	Quality Assurance.
WSDL	Web Services Description Language.
KSP	Kotlin Symbol Processing Api.
KAPT	Kotlin Annotation Processing Tool.

Bibliography

[AFT$^+$14] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE software*, 32(5):53–59, 2014.

[AO16] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[BC07] R. C. Bryce and C. J. Colbourn. The density algorithm for pairwise interaction testing. *Software Testing, Verification and Reliability*, 17(3):159–182, 2007.

[BC09] R. C. Bryce and C. J. Colbourn. A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification and Reliability*, 19(1):37–53, 2009.

[BHM$^+$14] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[BOF14] F. P. Basso, T. C. Oliveira, and K. Farias. Extending junit 4 with java annotations and reflection to test variant model transformation assets. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1601–1608, 2014.

[BSSH17] A. Bainczyk, A. Schieweck, B. Steffen, and F. Howar. Model-based testing without models: the todomvc case study. *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, pages 125–144, 2017.

[CBGS18] M. Camilli, C. Bellettini, A. Gargantini, and P. Scandurra. Online model-based testing under uncertainty. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 36–46. IEEE, 2018.

[CDFP97] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[CDKP94] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. The automatic efficient test generator (aetg) system. In *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*, pages 303–309. IEEE, 1994.

[CDS07] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139, 2007.

[CG09] A. Calvagna and A. Gargantini. Ipo-s: incremental generation of combinatorial interaction test data based on symmetries of covering arrays. In *2009 International conference on software testing, verification, and validation workshops*, pages 10–18. IEEE, 2009.

[Col08] C. J. Colbourn. Covering array tables for t=2,3,4,5,6, 2008.

[Col18] Open Source Collective. Open api generator. `https://github.com/OpenAPITools/openapi-generator/blob/master/docs/generators/kotlin.md`, 2018. [Online].

[CV14] W. Cazzola and E. Vacchi. @ java: Bringing a richer annotation model to java. *Computer Languages, Systems & Structures*, 40(1):2–18, 2014.

[Cze06] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, pages 419–430. Citeseer, 2006.

[Dar09] I. Darwin. Annabot: A static verifier for java annotation usage. *Advances in Software Engineering*, 2010, 2009.

[Dav19] A. L. Davis. Gradle. *Learning Groovy 3: Java-Based Dynamic Scripting*, pages 105–114, 2019.

[DJK$^+$99] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, pages 285–294, 1999.

[doy21] doyaaaaaken. Kotlin csv library. `https://github.com/doyaaaaaken/kotlin-csv`, 2021. [Online].

[EWZC15] V. Entin, M. Winder, B. Zhang, and A. Claus. A process to increase the model quality in the context of model-based testing. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–7. IEEE, 2015.

[FBJ16] K. Frajták, M. Bures, and I. Jelinek. Model-based testing and exploratory testing: Is synergy possible? In *2016 6th International Conference on IT Convergence and Security (ICITCS)*, pages 1–6. IEEE, 2016.

[GCD09] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *2009 1st International Symposium on Search Based Software Engineering*, pages 13–22. IEEE, 2009.

[GCD11] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16:61–102, 2011.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification, 3rd Edition*. Addison-Wesley Professional, 2005.

[GLL+21] B. Garn, D. Sebastian Lang, M. Leithner, D. Richard Kuhn, R. Kacker, and D. E. Simos. *Combinatorially XSSing Web Application Firewalls*. 2021.

[GO07] M. Grindal and J. Offutt. Input parameter modeling for combination strategies. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pages 255–260. ACTA Press, 2007.

[Goo21] Google. Ksp- kotlin symbol processing api. `https://kotlinlang.org/docs/ksp-overview.html`, 2021. [Online].

[GR18a] A. Gargantini and M. Radavelli. Ctwedge: Combinatorial testing web-based editor and generator. `https://foselab.unibg.it/ctwedge/`, 2018. [Online].

[GR18b] A. Gargantini and M. Radavelli. Migrating combinatorial interaction test modeling and generation to the web. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 308–317. IEEE, 2018.

[GRG+19] B. Garn, M. Radavelli, A. Gargantini, M. Leithner, and D. E. Simos. A fault-driven combinatorial process for model evolution in xss vulnerability detection. In *Advances and Trends in Artificial Intelligence. From Theory to Practice: 32nd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2019, Graz, Austria, July 9-11, 2019, Proceedings 32*, pages 207–215. Springer, 2019.

[GS17] S. Gulati and R. Sharma. Java unit testing with junit 5. *Java Unit Testing with JUnit*, 2017.

[HGB08] B. Hasling, H. Goetz, and K. Beetz. Model based testing of system requirements using uml use case models. In *2008 1st international conference on software testing, verification, and validation*, pages 367–376. IEEE, 2008.

[HL05] R. Heckel and M. Lohmann. Towards contract-based testing of web services. *Electronic Notes in Theoretical Computer Science*, 116:145–156, 2005.

[HNS09] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *2009 ieee international symposium on parallel & distributed processing*, pages 1–11. IEEE, 2009.

[Ini11] Open Api Initiative. Open api. `https://www.openapis.org/`, 2011. [Online].

[Jet17] Jetbrains. Kotlin annotation processing tool. `https://kotlinlang.org/docs/kapt.html`, 2017. [Online].

[KBD$^+$15] D. R. Kuhn, R. Bryce, F. Duan, L. S. Ghandehari, Y. Lei, and R. N. Kacker. *Combinatorial Testing: Theory and Practice*, volume 99 of *Advances in Computers*. 2015.

[KLK08] R. Kuhn, Y. Lei, and R. Kacker. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10(3):19–23, 2008.

[KLS20] L. Kampel, M. Leithner, and D. E. Simos. Sliced aetg: a memory-efficient variant of the aetg covering array generation algorithm. *Optimization Letters*, 14:1543–1556, 2020.

[kot18] kotcity. City simulation written in kotlin. `https://github.com/kotcity/kotcity`, 2018. [Online].

[KP11] V. V. Kuliamin and A. A. Petukhov. A survey of methods for constructing covering arrays. *Programming and Computer Software*, 37:121–146, 2011.

[KS17] K. Kleine and D. E. Simos. Coveringcerts: Combinatorial methods for x. 509 certificate testing. In *2017 IEEE International conference on software testing, verification and validation (ICST)*, pages 69–79. IEEE, 2017.

[KS19] L. Kampel and D. E. Simos. A survey on the state of the art of complexity problems for covering arrays. *Theoretical Computer Science*, 800:107–124, 2019.

[LGS21] M. Leithner, B. Garn, and D. E. Simos. Hydra: Feedback-driven black-box exploitation of injection vulnerabilities. *Information and Software Technology*, 140:106703, 2021.

[LKK$^+$07] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. 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)*, pages 549–556. IEEE, 2007.

[LT98] Y. Lei and K.-C. Tai. In-parameter-order: A test generation strategy for pairwise testing. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*, pages 254–261. IEEE, 1998.

[Mar05] P. Marschall. Detecting the methods under test in java. *Bachelor thesis*, 2005.

[MBX07a] E. Martin, S. Basu, and T. Xie. Automated testing and response analysis ofweb services. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 647–654. IEEE, 2007.

[MBX07b] E. Martin, S. Basu, and T. Xie. Websob: A tool for robustness testing of web services. In *29th International Conference on Software Engineering (ICSE'07 Companion)*, pages 65–66. IEEE, 2007.

[MPS19] D. Maciel, A. CR. Paiva, and A. R. Da Silva. From requirements to automated acceptance tests of interactive apps: An integrated model-based testing approach. In *ENASE*, pages 265–272, 2019.

[MT19] A. Møller and M. T. Torp. Model-based testing of breaking changes in node. js libraries. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 409–419, 2019.

[ND08] C. Noguera and L. Duchien. Annotation framework validation using domain models. In *Model Driven Architecture–Foundations and Applications: 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings 4*, pages 48–62. Springer, 2008.

[Ora14] Oracle. Predefined java annotations. `https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html`, 2014. [Online].

[Paw06] R. Pawlak. Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11):1–1, 2006.

[PE07] C. Pacheco and M.D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[PJ09] V. K. Proulx and W. Jossey. Unit test support for java via reflection and annotations. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 49–56, 2009.

[PN15] P. Pigula and M. Nosal. Unified compile-time and runtime java annotation processing. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 965–975. IEEE, 2015.

[Raa] F. Raab. *System Under Test*, pages 1663–1665.

[Rei97] S. C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Proceedings Fourth International Software Metrics Symposium*, pages 64–73. IEEE, 1997.

[RV11] H. Rocha and M. T. Valente. How annotations are used in java: An empirical study. In *SEKE*, pages 426–431, 2011.

[SGZL19] D. E. Simos, B. Garn, J. Zivanovic, and M. Leithner. Practical combinatorial testing for xss detection using locally optimized attack models. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 122–130, 2019.

[SL89] D. P. Sidhu and T.-K. Leung. Formal methods for protocol testing: A detailed study. *IEEE transactions on software engineering*, 15(4):413–426, 1989.

[SPBL13] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto. Security applications of formal language theory. *IEEE Systems Journal*, 7(3):489–500, 2013.

[SZL19] D. E. Simos, J. Zivanovic, and M. Leithner. Automated combinatorial testing for detecting sql vulnerabilities in web applications. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, pages 55–61, 2019.

[TPW+02] W-T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang. Extending wsdl to facilitate web services testing. In *7th IEEE International Symposium on High Assurance Systems Engineering, 2002. Proceedings.*, pages 171–172. IEEE, 2002.

[WKS20a] M. Wagner, L. Kampel, and D. E. Simos. Ipo-q: a quantum-inspired approach to the ipo strategy used in ca generation. In *Mathematical Aspects of Computer and Information Sciences: 8th International Conference, MACIS 2019, Gebze, Turkey, November 13–15, 2019, Revised Selected Papers 8*, pages 313–323. Springer, 2020.

[WKS+20b] M. Wagner, K. Kleine, D. E. Simos, R. Kuhn, and R. Kacker. Cagen- covering array generation. `https://matris.sba-research.org/tools/cagen`, 2020. [Online].

[WKS+20c] M. Wagner, K. Kleine, D. E. Simos, R. Kuhn, and R. Kacker. Cagen: A fast combinatorial test generation tool with support for constraints and higher-index arrays. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 191–200. IEEE, 2020.

[WKS21] M. Wagner, L. Kampel, and D. E. Simos. Heuristically enhanced ipo algorithms for covering array generation. In *Combinatorial Algorithms: 32nd International Workshop, IWOCA 2021, Ottawa, ON, Canada, July 5–7, 2021, Proceedings 32*, pages 571–586. Springer, 2021.

[YDL+15] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn. Constraint handling in combinatorial test generation using forbidden tuples. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–9. IEEE, 2015.

[YLKK13] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn. Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 370–375. IEEE, 2013.