TAF: a tool for diverse and constrained test case generation

Clément Robert, Jérémie Guiochet, Hélène Waeselynck, Luca Vittorio Sartori

TAF: a tool for diverse and constrained test case generation

Clément Robert, Jérémie Guiochet, Hélène Waeselynck, Luca Vittorio Sartori
LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France
Email: firstname.lastname@laas.fr

***Abstract*—The generation of test cases may have to accommodate size-varying data structures and semantic constraints between the data elements. This often requires the development of custom generators. In this paper, we introduce a novel generic tool to generate constrained and diverse test cases from a data model. First, the user defines the model using an XML-based domain-specific language. Then TAF generates diverse test cases by combining random sampling with the use of an SMT solver. The capabilities of the tool are demonstrated by four examples of models coming from various application domains: virtual crop fields for testing an agriculture robot, bitmap images with a graduated background, a population of taxpayers in a tax management system, and tree structures of diverse sizes and heights. We show how TAF performs in terms of data diversity and execution time. We also provide some comparison results with an UML-based tool using SMT solving.**

***Index Terms*—test, test input generation, fuzzing, autonomous robots, simulation**

I. Introduction

Many applications require test cases with rich and structured data content. For example, the starting point of this work was the simulation-based testing of mobile robots. In this context, a test case includes the description of the 3D-scene in which the robot is placed. The scene is a data structure composed of elements of various types, the number of elements as well as their attributes having to satisfy some semantic constraints. Beyond this example, size-varying data structures and inter-dependent data elements are quite common for information processing systems, communication protocols, etc.

The production of synthetic test data for such systems may involve custom test generators, but this paper investigates a more generic solution based on a data model. The candidate approach should: (i) require no user effort other than the declarative specification of the data, (ii) accommodate rich data structures and numerical properties, (ii) provide diversity in the generated data contents and size.

We investigated the numerous existing tools for test generation. Surprisingly, we did not find any tool that could match the three objectives at the same time. In particular, the approaches using constraint solving seemed promising. They fit well with declarative models. A number of mature solvers are available off-the-shelf, including SMT (Satisfiability Modulo Theories) solvers that can handle numerical constraints. However, they fail short of our objectives for two reasons. First, the backend solvers work in a fixed universe, i.e., with a fixed set of variables and constraints. They offer little support for exploring size-varying data structures, when the number and type of contained elements is not fixed, and the creation of data instances yields an evolving set of variables and relations. Second, the solvers are deterministic and return one solution, which is (in some way) the most obvious one they find to satisfy all constraints. This is not satisfactory for test generation, as one would like to generate multiple test cases and obtain diversity in the coverage of the valid data space.

This paper presents the solution we designed to address these issues. It proposes a novel way to harness the constraint solving, which ensures diverse data contents and size. The harness dynamically updates the constraints as data instances are generated, and combines random sampling with constraints solving to enforce diversity. In particular, the generation injects random values into the queries sent to the solver.

TAF (Test Automation Framework) [1] prototypes this approach on top of the Z3 SMT solver [2]. The user specifies a data model in an XML-based language. The tool then offers data generation, completion, and export facilities. We demonstrate the capabilities of the tool by four case studies exemplifying different types of data models: virtual crop fields for testing an agriculture robot, bitmap images with a graduated background, tree data structures of diverse sizes and heights, and a population of taxpayers in a tax management system.

The structure of the paper is as follows. Section II presents related work on test case generation. Section III gives an overview of the tool, and introduces our motivating example. Our test case description language is detailed in Section IV, then Section V presents the generation algorithms. Section VI shows the use of the tool on four case studies. We also provide some comparison results with an alternative tool. Finally, Section VII concludes.

II. Related work

We discuss related work that addresses the generation of test cases from rich input data models. Here, we do not consider approaches that derive the test data from the source code. Rather, we focus on black box approaches for system-level testing or when the source code is not available. The

The research presented in this paper has been partly funded by European Union's EU Framework Programme for Research and Innovation Horizon 2020 under Grant Agreement No. 812.788 (Safer Autonomous Systems).

generation proceeds from data models expressed in XML-based languages [3], [4], first-order languages [5], [6], formal grammars [7]–[14], UML [15]–[17], or even general-purpose languages [18]–[20].

As soon as the data model involves semantic properties, the production of valid instances becomes an issue. We classify the generation approaches in three broad categories, depending on whether they rely on many successive trials, or they tune the generation process for the specific model under consideration, or they delegate the generation to constraint solvers.

A. Generate-then-filter approaches

In practice, many approaches implement a form of the generate-then-filter algorithm. It consists in producing a candidate data instance, checking it, and discarding it if it is semantically invalid. The process is repeated.

The property-based approaches popularized by QuickCheck [18] work according to this principle: they randomly generate inputs and discard the ones that do not fulfill the precondition of the property to check. Another example is Yagg [10], a generation framework based on attribute grammars. Yagg uses the context-free part of the grammar to produce the candidate test cases, and then apply the context-sensitive checks to filter out the invalid ones.

Our framework TAF addresses generation problems where such an approach is inefficient, and yields a high rate of invalid candidate test cases.

B. Approaches that tune the generation process

The rate of invalid cases can be decreased by tuning the generation process. It requires the introduction of application-specific knowledge on how to build the data.

Several approaches annotate the data model with user-supplied code fragments. The commercial fuzzer Peach [4] (now integrated in DevSecOps) has an XML-based language with constructs for inserting references to external functions. This provides a means to aid in the generation of valid data at the expense of some coding effort. Similarly, in the framework of formal grammars, Maurer [9] attaches *Action routines* to the production rules. Part of the data may then be generated based on the context. The grammar annotation scheme by Kifetew et al. [14] manages a context that carries information about the data types, which proved helpful to build valid integer or float expressions. Some authors have used higher-order attribute grammars (HAGs), for which the separation between the semantic attributes and the syntactic constructs disappears. The mutation-based fuzzer in [12] leverages this facility: the user-supplied annotations can manipulate the syntax tree of the test cases to keep the data valid after a mutation.

For weighted grammars and other stochastic models, an alternative way to improve the generation process is by tuning the probabilities of the choices. In [20], these probabilities are optimized by search-based techniques. In [14], they are learned from an existing pool of valid test cases. Note that the latter work also proposes a grammar annotation scheme (mentioned in the previous paragraph), which produces a higher number of valid data than the learned probabilities. In [13], a grammar rule can change the probability of another one, providing yet another scheme to manually adapt the choices to the context.

All these approaches require an effort from the user to customize the generator. The effort is to be done for each new application. In contrast, our framework TAF offers a generic solution that works out of the box. The user only has to specify the data structures and the desired properties. The model is kept declarative and free from programmatic annotations.

C. Approaches that use constraint solving

A solution to spare the tuning effort is to delegate the construction of valid test cases to off-the-shelf constraint solvers (as we do in TAF).

In the context of SAT solving, Alloy [21], [22] is worth mentioning. The core first-order language is well suited for expressing structure-rich models. The analyzer is able to create model instances, making it usable for test generation. For example, TestEra [5] is a framework for testing Java programs based on Alloy. It uses the Alloy analyzer to enumerate the set of all non-isomorphic valid inputs up to a certain size. Note that, in this case, the aim is to perform bounded exhaustive testing, which induces a focus on small data sizes. Also, since Alloy is connected with SAT-based analysis, the approach does not support numerical constraints over real variables.

UMLtoCSP [16] uses constraint logic programming (CLP) to analyze UML models including constraints in OCL (Object Constraint Language [23]). It supports numerical constraints. Like Alloy, the framework is able to create model instances with a bounded size. The authors recommend that the size be kept small. Dewey et al. [24] demonstrate the efficiency of CLP to generate data structures, but observe that CLP specifications may become obscure. They recommend the design of small frontend languages with loops and variable assignments, for which an efficient translation to CLP would be investigated.

Other approaches use SMT solving to address UML models with numerical constraints. Cantenot et al. [15] generate tests in a fixed universe: the user has to predefine the desired class instances in an object model. The PLEDGE framework [17] directly generates the class instances and their contents from the UML class diagram. It addresses the lack of scalability of constraint solvers by hybridizing metaheuristic search and SMT. The authors show that this strategy is able to create larger model instances than Alloy and UMLtoCSP. The hybrid strategy also outperforms the pure search-based one using the metaheuristic only.

Among all generation approaches, PLEDGE is the closest one to ours since TAF hybridizes random sampling and SMT. The hybridization algorithms are however different. PLEDGE has randomness in the creation of class instances but does not attempt to ensure diversity of their contents. It takes the solution returned by the SMT solver. TAF has randomness in both the instantiation process and the production of data contents, with the aim of maximizing diversity. It forces the solver to produce different solutions.

To the best of our knowledge, this contribution is novel. As noted by Dutra et al. [6], the generation of multiple diverse solutions has been little studied for SMT constraints. The authors have addressed the problem for theories of bit vectors and bit-vector arrays used in hardware verification. Their approach relies on combinations of bitflips to explore the diverse solutions in a fixed universe. With TAF, we propose another approach targeted at size-varying data structures with numerical constraints. In addition to providing diversity, the approach is also efficient for creating data instances of a realistic size, as will show the evaluation against PLEDGE in Section VI.

III. Framework overview

In order to illustrate the facilities offered by the TAF framework, let us first provide a motivating example inspired by [25]. It concerns the simulation-based testing of an agriculture robot, an autonomous weeder that operates in fields of vegetables. A test case for this system consists of a weeding mission in a virtual crop field (see Figure 1). A crop field is composed of an arbitrary number of crop rows. The robot has to navigate along them. When it arrives at the end of a row, it makes a U-turn to weed the next one. We assume a monoculture environment, hence the field contains a single type of vegetables, which can be cabbages or leeks. One of the mission parameters, *is_first_track_outer*, indicates whether or not the robot starts at the external side of the field.

The generation of such virtual test cases can proceed in two steps. The first step generates abstract cases from a data model, while the second one produces the concrete test artifacts to be fed into the test platform. By having these two separate steps, it is possible to keep the modeling focused on high-level data elements, without being overwhelmed by the complexity of concrete details or formatting issues. For example, an abstract test case of the autonomous weeder only specifies a number of crop rows and a vegetable type. It is then concretized by creating a scene description file in a format understood by the simulator, e.g., in terms of a list of vegetable meshes to be placed at designated coordinates.

The TAF framework supports this test generation process (see Figure 2). It provides an XML-based language to specify *test templates*, i.e., structured models of the data composing an abstract test case. The *data factory* generates abstract test cases according to the template. To facilitate the final concretization step, TAF also creates a *code skeleton*, in which export functions are attached to the various data elements. These export functions are initially empty, the user should add the custom export code in order to convert the produced test cases (in XML) into the required file format.

A salient feature of TAF is its possibility to accommodate semantic constraints. In the motivating example, the various data elements cannot take independent values. If the crop field has only one row, then the weeding mission necessarily starts at an external side of the field. Consecutive rows must have nearly the same length, say by +/- 10%. We furthermore require that the lengths of the first and last row are also close by +/- 10%. Such constraints can be expressed into the template and make the test generation challenging. On the one hand, a random generate-then-filter strategy would be inefficient. Suppose we have tens of rows: it would take a huge number of trials before obtaining a set of consistent row lengths. On the other hand, the use of a constraint solver would return a single solution, possibly the simplest one, like all rows having the same length, and this length is the minimal one allowed for a row. To avoid both pitfalls, the data factory combines random generation with constraint solving, allowing for the efficient production of diverse test cases that satisfy the constraints. The framework offers tunable random generation functions for each basic type of data, and uses the SMT solver Z3 for the satisfaction of constraints.

Another convenient facility is the possibility for the user to supply a *partially instantiated test case*. This input file is optional. It allows for setting the values of any subset of the basic data elements. For example, the user can request a crop field with a maximal number of rows, or request the combination of this number of rows with a cabbage type of vegetables. The data factory will generate the missing data elements to produce a complete and valid test case. This test case completion facility gives great flexibility to explore subsets of the input space or cover extreme cases. Moreover, the choice of XML-based file formats is intended to facilitate the implementation of test strategies on top of TAF. Both the template and partial instance files can be manipulated to request specific values, add or remove constraints, or change the default generation function for a given data element.

The following sections provide more details about the data factory: the XML-based description language it accepts, and its generation algorithm.

IV. Test case description language

This section describes the language used in the XML files *Template* and *Partial instance* from Figure 2. It features a markup-based declaration of the data elements and embeds constraints. The language was designed with the ontology presented in Figure 3. The .dtd and the BNF are provided on the Git repository [1]. In order to illustrate the main concepts, we will use the template file in Figure 4 that gives the data model of our motivating example. We first present how the data structure is declared, from composite elements to parameters having a basic type. We also explain how the parameters are attached a default generator. Then, we introduce the expression of constraints between parameters. Finally, we present how to assign values to parameters in a partially instantiated test case.

A. Data structure

As shown in Figure 3, a test case template involves four different types of XML elements: $Root$, $Node$, $Parameter$, and $Constraint$. Every element must have a "name" attribute. Elements are nested to form a tree starting from the root. The root and nodes are composite data structures with child

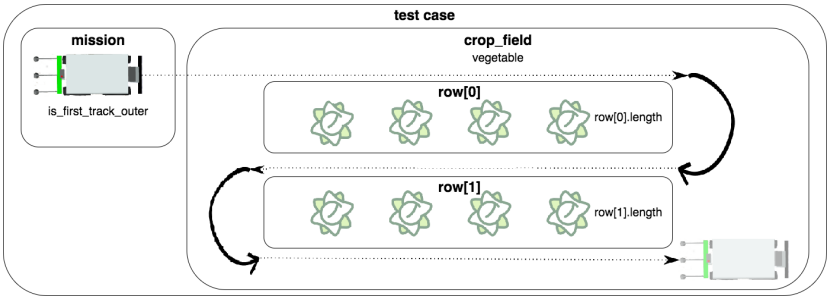

Fig. 1. A virtual test case for the autonomous weeder

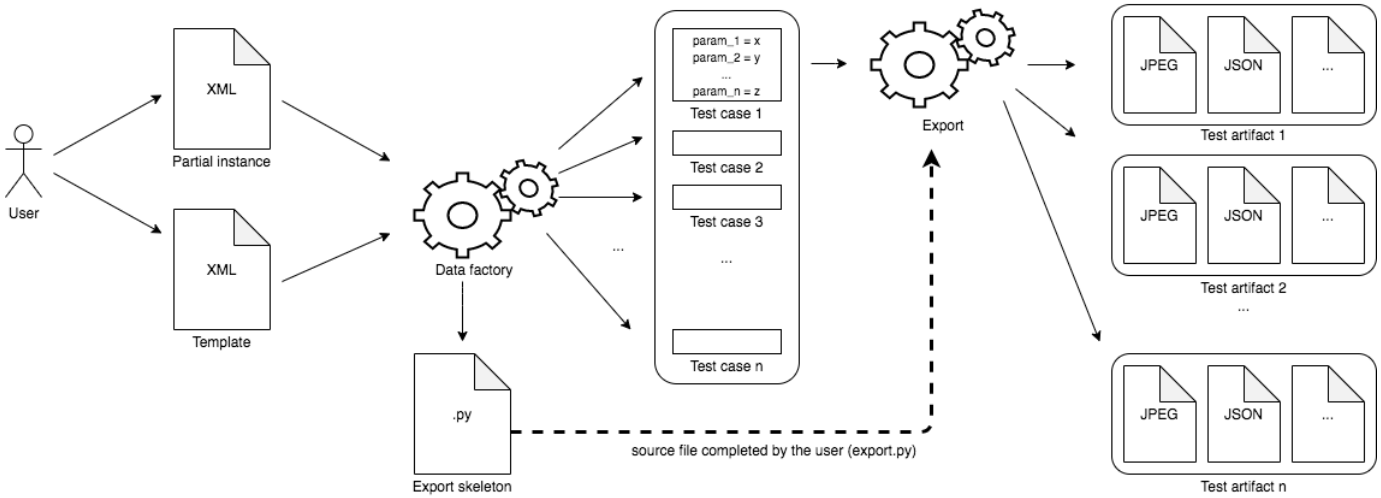

Fig. 2. Overview of the TAF framework

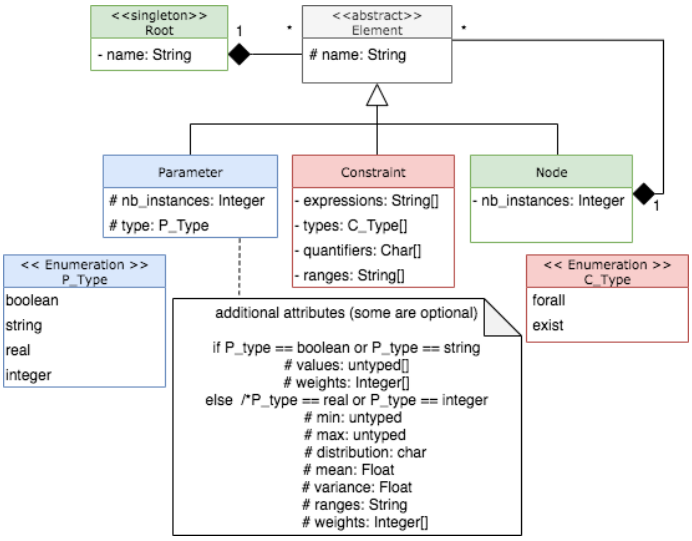

Fig. 3. TAF DSL ontology with a UML class diagram

elements, while a parameter is not composite. The types of parameters (P_Type) are boolean, string, real, and integer.

The root is unique and mandatory, but both the parameters and nodes have a "nb_instances" (–number of instances–) meta-attribute that allows for multiplicity. If multiplicity is not explicitly declared in the template, the number of instances is supposed to be 1.

The template in Figure 4 illustrates these structural concepts. The test case root (Line 3) is composed of a node "field" (L4) and a node "mission" (L19). The node "field" has one instance (L4), and contains a parameter "vegetable" (L5) that can take the values "leek" or "cabbage". A field is composed of multiple rows. In the declaration of the "row" node (L6), the allowed number of instances is specified by its min and max values (1 and 40). Each row element contains a parameter "length" (L7), with min and max values as well (10 and 100). As a general rule, all numerical parameters (real or integer) must have an explicit definition range, and all string parameters must have a set of candidate values.

B. Default generators

TAF attaches a default generator to each parameter in the structure. Its aim is to produce diverse values from the parameter type. If nothing is specified in the template, uniform sampling over all possible values is used as the default. For instance, in Figure 4, L7, the parameter "length" will be determined in the range $[10, 100]$ with a uniform sampling. But the user also has the possibility to select other default generators. This is done when the parameter is declared, by

```
1  <?xml version="1.0"?>
2
3  <root name="test_case">
4      <node name="field" nb_instances="1">
5          <parameter name="vegetable" type="string" values="cabbage;leek" weights="5;7"/>
6          <node name="row" min="1" max="40">
7              <parameter name="length" type="real" min="10.0" max="100.0"/>
8              <constraint name="interval" types="forall"
9                          expressions="row[i]\length INFEQ 1.1*row[i-1]\length;
10                                      row[i]\length SUPEQ 0.9*row[i-1]\length"
11                         quantifiers="i"
12                         ranges="[1, row.nb_instances-1]"/>
13             <constraint name="interval_2"
14                         expressions="row[0]\length INFEQ 1.1*row[row.nb_instances - 1]\length;
15                                      row[0]\length SUPEQ 0.9*row[row.nb_instances - 1]\length"/>
16         </node>
17     </node>
18
19     <node name="mission" nb_instances="1">
20         <parameter name="is_first_track_outer" type="boolean"/>
21         <constraint name="first_track"
22                     expressions="IMPLIES(..\field\row.nb_instances EQ 1, .\is_first_track_outer EQ True)"/>
23     </node>
24 </root>
```

Fig. 4. Template file example for the autonomous weeder simulation

using dedicated attributes (e.g., the weights attribute appearing in the note on the Parameter class of Figure 3).

The set of available generators depends on the data type. Boolean and string parameters can have weighted choices. For instance, in Figure 4, L5, the declaration of the vegetable parameter introduces a biased sampling of values, where the choice "leek" (of weight 7) is more likely than "cabbage" (of weight 5). It is required to have as many weights as values. For integer or real parameters, there are two alternatives to uniform sampling over their definition range. The user can assign weights to subranges of values, or request sampling according to a normal distribution with some mean and variance. The parser of the template will check that the requested generator is compatible with the parameter type.

C. Constraints

Constraints consist of expressions that specify semantic properties in a descriptive way. The language syntax lets the user specify a list of one or more expressions separated by a semi-colon (Figure 4, L8, L13, L21).

The expressions may involve logical (not, and, or, implies), arithmetic (+, -, *, /), and relational (==, !=, <, <=, >, >=) operators. The concrete syntax ensures that the file can be parsed by any regular XML parser. In particular, the relational operators are written as EQ, DIF, INF, INFEQ, SUP, SUPEQ to avoid issues with characters < and >. For easier interfacing with Z3, the structure of the expressions follows the Z3 syntax (e.g., for expressions with logical operators as in L22 of Figure 4, the operator is first, and the operands come after).

The variables can be any parameter of the test case structure. They are referenced by an access path relative to the location where the constraint is declared. The path syntax uses the windows file system notation with separators "\", to avoid ambiguity with the division symbol. Moreover, "." and ".." refer to the current node and the parent node. Paths can include indices to refer to the instances of nodes. In Figure 4, L9, $row[i]\backslash length$ refers to the length parameter of the ith row.

Our language also provides quantification over finite structures. For instance, the constraint named "interval" (L8-12) has a universal quantification ($forall$) over all row instances. It has a single quantified variable (i in L11) taking the value of row indices (L12). The language also provides existential ($exist$) quantification (not used in our motivating example). Note that it is possible to use nested quantifiers of universal and existential types. An example is given in the tax payer template on the git repository in the "templates" directory [1].

D. Partially instantiated test cases

An important feature is the possibility to feed partially instantiated test cases into TAF. If possible, the test case is completed according to the template model.

The partial test case file has a structured description of a subset of element instances. Some parameters are forced to a desired value. They appear with their name and a value attribute. Figure 5 shows an example where the length of the third row (with row numbering starting at 0) is forced to 40. Accordingly, TAF will consider that the template is augmented with two constraints: that the number of the row instances is > 2, and that the third row has a specific length. If the desired length is out-of-range, the parsing will detect it and abort the generation. For the convenience of the user, predefined keywords facilitate the assignment of some values of interest (e.g., "max" for the maximal value of a numerical parameter, "first" for the first possible value of a string parameter).

Note that the user who creates a partially instantiated test case is not necessarily human. A program may produce the partial instances according to some test strategy, and rely on TAF to complete them.

```
<?xml version="1.0"?>

<root name="test_case">
    <node name="field">
        <node name="row" instance="2">
            <parameter name="length" values="40"/>
        </node>
    </node>
</root>
```

Fig. 5. Partially instantiated test case

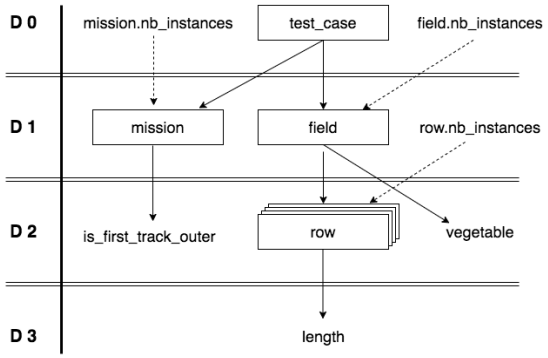

Fig. 6. Layered generation approach

V. Generation algorithms

The generation adopts a top-down, layered approach that follows the structure of the XML template. The parsing of the template file builds an initial tree. It contains placeholders for the elements defined in the template according to the containment hierarchy. As the generation proceeds, the tree is updated to insert the created element instances, and assign values to their parameters.

Figure 6 illustrates the top-down approach on our running example. The root test case node is at depth 0 of the tree, its mission and crop_field children at depth 1, and so on. All elements at a given depth d are generated at the same step, after the elements at depth $d-1$ and before the ones at $d+1$. The number of instances to create at each layer is a parameter *<element_name>.nb_instances* that belongs to the immediately upper layer, the one of the parent node. It guarantees that the number of instances is always defined before building the related objects. If the template lets this parameter implicit, it takes the default value one. In the running example, the number of mission and crop_field instances is set to 1, hence we build one instance of each of them. In contrast, the template specifies any number of rows in a $[min, max]$ range: a number will be chosen at depth 1, when the parent crop_field node is processed.

Each constraint defined in the template is also assigned a depth: it takes the maximal depth of all the parameters that it refers to. In the running example, there is a constraint between $row.nb\_instances$ and $is\_first\_track\_outer$, respectively of depth 1 and 2. According to the previous rule, this is a depth 2 constraint. It means that $row.nb\_instances$ will be generated first, and later the generation will try to find a value of $is\_first\_track\_outer$ that satisfies the constraint. This approach can obviously lead us to a dead-end as the parameters generated from the previous layers may produce unsolvable constraints. Hence the generation uses backtracking.

Algorithm 1 shows the layered generation with backtracking. The main loop (Lines 6-18) iterates until either the generation is complete or the maximal budget for backtracking is reached (MAX_B). This budget is configured by the user. At each iteration, function $generate\_depth()$ attempts the generation of the elements at the current depth. If it succeeds, the generation continues at the next depth (Line 8). If it fails, the choices made at the current depth are undone, and the generation goes back to the previous layer (Lines 11-13). After exiting the loop, the tool reports on its failure (Line 20) or success (Line 23). In the latter case, an XML version of the test case is created for archiving purposes.

Algorithm 1 main

```
Require: path
1: tree ← parse_template(path)
2: create_export_code_skeleton_file(tree)
3: preprocess(tree)
4: counter ← 0
5: depth ← 0
6: while counter < MAX_B AND depth ≤ tree.max_depth() do
7:     if generate_depth(tree, depth) then
8:         depth ← depth + 1
9:     else
10:        if depth > 0 then
11:            counter ← counter + 1
12:            depth ← depth − 1
13:            tree.reset(depth)
14:        else
15:            return False
16:        end if
17:    end if
18: end while
19: if counter == MAX_B then
20:    return False
21: end if
22: create_XML_TestCase_file(tree)
23: return True
```

Algorithm 2 describes how a given layer is generated. First, it extracts the parameters p and the constraints c of the current depth. Then, there is a triage between dependent and independent parameters, according to whether they are involved in any of the constraints (Lines 3-4). Independent parameters are simply generated using the default random generators (Lines 5-7). If there is no constraint, the generation of the layer is done (Lines 8-10). Otherwise, a $problem$ object is created with the dependent parameters and the constraints (Line 12). If the problem fails the satisfiability check in Z3, the layer is not generated and the function returns $False$ (Line 14). This will cause backtracking in the main algorithm. If the problem is satisfiable, the dependent parameters are processed. Their generation mixes constraint solving and randomness in order to ensure diversity in the solutions. Such a process is described in Algorithm 3 to be presented shortly. Finally, before returning success, the algorithm prepares the next layer according to the newly generated values of *<element_name>.nb_instances* parameters (Lines 9 or 17): the appropriate number of element nodes is created and inserted into the tree.

Algorithm 2 generate_depth

```
Require: tree, depth
 1: p ← extract_parameters(tree, depth)
 2: c ← extract_constraints(tree, depth)
 3: dependent_p ← filter_dp(p, c)
 4: independent_p ← filter_indp(p, c)
 5: for p' in independent_p do
 6:     p'.default_generate()
 7: end for
 8: if c is empty then
 9:     create_next_depth_instances(p)
10:     return True
11: end if
12: problem ← init_problem(dependent_p, c)
13: if NOTproblem.check_sat() then
14:     return False
15: else
16:     solve_with_diversity(problem)
17:     create_next_depth_instances(p)
18:     return True
19: end if
```

Algorithm 3 takes a problem object as input and solves it with diversity. We know that the problem is satisfiable (from the previous algorithm), but want to force the solver to produce a different solution than the one it would normally return. For this, the principle is to assign random values to as many parameters as possible before solving the constraints. In the inner loop at Lines 5-12, we keep adding new constraints $p = v$ until the problem becomes unsatisfiable. It yields a new solvable version of the problem where some parameters take random values. The process is repeated multiple times according to a diversity budget MAX_D (outer loop starting at Line 3). At each iteration, we keep track of the best problem in terms of the number of randomly chosen parameters (Line 14). Lastly, the best problem is solved, and all the parameters are updated accordingly (Lines 17-18).

Algorithm 3 solve_with_diversity

```
Require: problem
 1: current_best ← problem
 2: counter ← 0
 3: while counter < MAX_D do
 4:     new_problem ← copy(problem)
 5:     while new_problem.check_sat() do
 6:         p ← select_parameter(problem)
 7:         p.default_generate()
 8:         v ← p.value
 9:         p.reset()
10:         c ← build_constraint(p, v)
11:         new_problem.add_constraint(c)
12:     end while
13:     remove_last_constraint(new_problem)
14:     current_best ← choose_best(current_best, new_problem)
15:     counter ← counter + 1
16: end while
17: current_best.solve()
18: current_best.update_parameters()
```

The TAF test case factory implements the above algorithms in Python. For simplification purposes, the presentation has focused on the template-based generation, omitting the test completion facility. In practice, the submission of a partially instantiated test case only impacts the preprocessing of the tree structure in Algorithm 1. It adds new constraints to force the desired parameter values, then the layered generation proceeds as described. The user may also request the generation of N test cases, rather than just one. Both the backtracking and diversity budgets (MAX_B and MAX_D) can be easily configured, should the default setting be insufficient for the target generation problem.

VI. Case studies

In this section we demonstrate the use of TAF on four case studies coming from different application domains. We assess its performance and the data diversity it supplies. $Performance$ is measured by the execution time needed to generate 100 valid test cases from each of the four models. The generation is repeated 10 times, and the median, min, max times are reported. *Data diversity* is analyzed in terms of the coverage of the data space. For each case study, we manually derive a set of cases to be covered, considering both the raw definition domains of the parameters (e.g., a range decomposed into subranges to be covered) and some application-specific aspects related to the semantics of the data. We monitor the accumulated percentage of covered cases while the generation proceeds and identify the missing cases at the end.

TAF is also compared with two alternative strategies: generate-then-filter and PLEDGE [17]. From our analysis of related work (see Section II), PLEDGE emerges regarding its ability to handle size-varying data structures and numerical constraints. Moreover, the hybridization of metaheuristic search and SMT allows it to create larger data instances than the other solver-based approaches.

A. Data models of the four case studies

We used our XML-based language to model the following four case studies. The corresponding template files are available on the git [1].

Oz: This is the industrial case study of an agriculture robot [25], from which our motivating example is extracted. The complete model has 7 composite elements, 15 parameters, and 5 constraints. The structure is presented as a class diagram in Figure 7. Compared to the smaller motivating example, the few additional constraints are on the number of instances of the new elements. For example, the number of row spacing values (element *inner_track_width*) must be consistent with the number of rows. The motivating example reproduces the most challenging constraints in this case study (the ones on the row lengths) and already gives a good idea of what the complete Oz model is like in TAF.

Bitmap: The aim is to create a gradient image in the bitmap format. The darkest pixel is at the bottom left corner and the lightest one at the upper right (see Figure 8). The content of the image is structured as a set of pixel rows, where each pixel contains a grayscale value. A padding parameter determines the number of bytes to add after each row, so that the total number of bytes is a multiple of 4. The model is concise, having two composite elements, 2 parameters, and 4 constraints. Formatting issues are delegated to the export code, e.g., to produce the image header given the size of the data and a predefined encoding option. Note that the generation of the image content may be challenging for constraint solving, as each pixel instance depends on its upper and right neighbors.

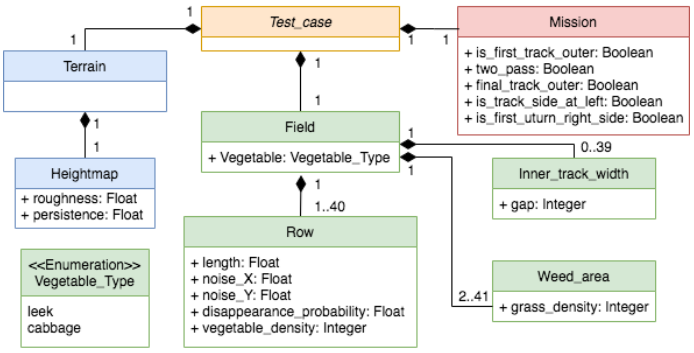

Fig. 7. Oz complete case study structure with a UML class diagram

Fig. 8. A gradient image created by TAF

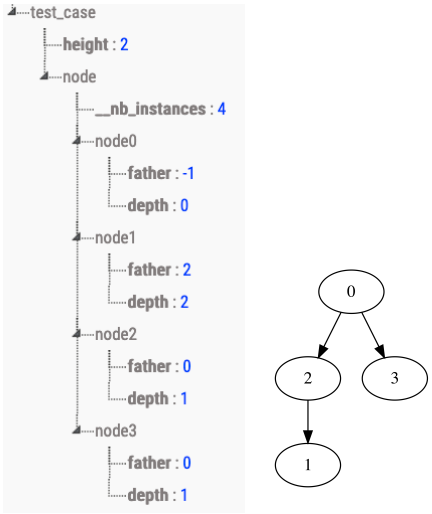

Fig. 9. A tree instance: XML encoding and graphical export

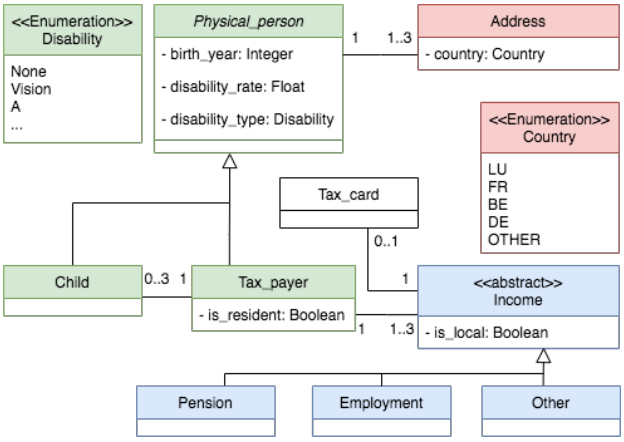

Fig. 10. UML of the tax payer case study

The value of the row padding is specified by a constraint involving modulo operations.

Tree: We study the generation of tree structures of diverse sizes and heights. TAF does not allow recursive definitions, hence it is not possible to model a tree as a composite element that would contain child trees. Rather, we have to consider a flattened set of tree nodes, and introduce the tree properties by quantification over this set. In our model, a node has two parameters: a $father$ (containing the id of the parent node) and a $depth$. Figure 9 shows the encoding (XML in a tree structure) of a tree instance of size 4 and height 2. Except for *node[0]*, which is the tree root, the other nodes may have an arbitrary position in the tree, as determined by the value of their parameters. We simply require that a node has the depth of its father +1, that no node has a depth greater that the global tree height, and that at least one node has a depth equal to the global height. These constraints remain abstract, not being prescriptive in how to generate the father relations consistently with the height. This may be challenging for the solver.

TaxPayer: This case study is the running example of the PLEDGE paper [17]. Its UML model is shown in Figures 10 and 11. It describes the data for an income tax management application. In this model, "Tax_payer" is a "Physical_person" that can support "Child" and earns 1 or more "Income". There are five OCL constraints (C1 to C5). For instance, C3 states that Taxpayers are not resident if they earn local incomes but do not have any address is Luxembourg. None of the OCL constraints is difficult to express in TAF. More difficult are object paradigms such as inheritance (e.g., "Child" inherits from "Physical_person") and bidirectional association (a "Physical_person" resides at an "Address"). Indeed, TAF only supports composition relations. We modeled the inheritance relation by "flattening" the inheritance (redistribute information in sub-classes). Our model is thus less concise than the original one, as some declarations and constraints are duplicated. Regarding the associations, as they were one-to many relations we were able to encode them as a uni-directional composition. A more generic solution would add parameters and encode symmetric indices in associated nodes. The TaxPayer model in TAF involves 9 composite elements, 12 parameters and 7 constraints.

The four case studies illustrate various types of data (from a matrix of pixels to Person elements), of structural relations (from hierarchical containers to other architectural relations), and of constraints (from simple logical constraints to more complex ones with nested quantification and arithmetic expressions). In addition to modeling them in the XML-based language, we developed external programs to check the valid-

```
context Physical_person inv C1:
self.birth_year >=1920 and self.birth_year <2020

context Physical_person inv C2:
if (self.disability_type = Disability::None)
then (self.disability_rate = 0)
else (self.disability_rate > 0 and
self.disability_rate <= 1.0) endif

context Tax_payer inv C3:
not self.address->forAll(a:Address |
        a.country <> Country::LU)
implies self.is_resident=true

context Tax_payer inv C4:
self.income->exists(inc:Income | inc.is_local=true) and
not self.address->exists(a:Address |
        a.country = Country::LU)
implies self.is_resident=false

context Income inv C5:
if(self.oclIsTypeOf(Other)) then
self.tax_card->size()=0 else
self.tax_card->size()=1 endif
```

Fig. 11. OCL code of the tax payer case study

ity of the generated data samples in each case. The checkers allowed us to confirm that the models encoded the intended properties and that the generation process was not flawed.

B. Performance

The performance of TAF heavily relies on the performance of the used solver. On the positive side, the solver processes only parts of the data. The layered approach decomposes the generation problem into subproblems. Furthermore, in a layer, only the dependent parameters are to be solved. On the negative side, the search for diversity involves multiple calls to the solver, which may be very costly. A question is whether the approach is practical with a state-of-the-art solver like Z3.

Table I shows the generation time of 100 test cases for each case study. The experiments are done on a 2017 MacBookPro14,2 dual-core (3.5 GHz Intel Core i7, RAM 16 GB). The backtracking and diversity budgets are both set to 10. The size of the generated data may vary in each test case and has the upper bound indicated in the second column of the table. For example, for Oz, the number of crop rows has its definition domain in 1..100. We stop a run if it is longer than two hours.

Tree is the most challenging case study for the solver. It failed to produce the 100 test cases within two hours when the tree size is up to 100 nodes. In the layered generation approach, the global height is decided first, and then the local decisions on node parameters have to comply with it. From our analysis, the long solving times are when the randomly injected father and depth values yield an unsat problem, but the solver does not detect this and keeps trying possibilities. By taking a max size of 50 nodes, the generation is feasible in less than 50 minutes. The easiest case studies are Oz and TaxPayer, for which 100 test cases are produced with a median time of respectively 17.62 and 133.9 seconds. For Oz, the data size is realistic since the crop fields in which the real robot operates usually have less than 100 rows. As regards TaxPayer, the loose coupling between person elements makes it possible

TABLE I
GENERATION TIME USING TAF (10 RUNS, 100 TEST CASES PER RUN)

Case study	Max data size	Tmin	Tmax	Tmedian
Oz	100 rows	16.36 s	18.79 s	17.62 s
Bitmap	100x100 px	636.7 s	1554 s	746.6 s
Tree	100 nodes	>2 h	>2 h	>2 h
	50 nodes	1961 s	2995 s	2664 s
TaxPayer	100 taxp.	123.9 s	161.9 s	133.9 s

TABLE II
GENERATION TIME USING PLEDGE (10 RUNS, 100 TEST CASES PER RUN)

Case study	Max data size	Tmin	Tmax	Tmedian
Oz	100 rows	1073 s	1327 s	1108 s
Tax payer	100 taxp	>2 h	>2 h	>2 h
	50 taxp	1640 s	2651 s	1804 s

to obtain a large population by generating multiple smaller populations and merging them together.

For comparison purposes, we considered the performance of two baseline strategies. The first one is generate-then-filter. We removed the semantic constraints from the data models, and used the external validity checker to filter the data generated by TAF. A first 2-hours run failed to produce any valid data except in the case of TaxPayer, which produced 46 test cases (out of 33,290 trials). Additional runs for TaxPayer never managed to obtain 100 valid cases in two hours (the best run obtained 55 cases). Moreover, all valid cases merely consisted of data instances with a single taxpayer. The generate-then-filter strategy is thus highly inefficient for our case studies. The second baseline is PLEDGE [17] which represents a more complex strategy similar to ours. PLEDGE uses the same backend solver as TAF (i.e., Z3) but harnesses it in a different way. The generation strategy combines metaheuristic search and SMT with the aim to use each technique where it is the most efficient. It distributes tasks to one technique or the other. In contrast, TAF has an intertwined usage of random sampling and SMT. We modeled Oz and TaxPayer in UML to supply them to PLEDGE. Table II shows the corresponding generation times. They were an order of magnitude longer than with TAF, even when the size of the data was reduced (TaxPayer).

To conclude, the Z3 solver is efficient enough for the proposed solve-with-diversity approach to be practical. Despite the multiple calls to the solver, the time for generating samples of medium-sized constrained data may be workable. The strategy allowed us to address four case studies for which a simpler generate-then-filter strategy fails. The comparison with the state-of-art tool PLEDGE shows that TAF can be very competitive in terms of performance.

C. Diversity of the test cases

We analyze the data space coverage provided by TAF. In the models, each numerical parameter or number of instances has a min-max definition range: we systematically split it into three subranges to cover, yielding low, medium and high cases. We

count a case as covered if any instance of the parameter has a value in the subrange. When a parameter has an enumerated type, we require the coverage of each value individually. We then add cases derived from the constraints and other semantic concerns, that address the relations between parameters. These cases are presented below.

Oz: The relative length of consecutive rows has to cover the +/-10% range split into low, medium and high cases. The same is required for the constraint between extremal rows, but we count coverage only if there are strictly more than two rows (i.e., if the first and last rows are not consecutive). We further require that the first row takes diverse values in its definition range.

Bitmap: The difference between a pixel value and the one of its upper (resp. right) neighbor has to range over 0..255, yielding low, medium and high cases. We also analyze the variation in an entire row or column of pixels, measuring the difference between the first and last pixel. Finally, we require that the darkest (resp. lightest) pixels takes diverse values in the images.

Tree: We require diversity in the ratio of the height to the number of nodes, to span the cases from a shallow tree with all nodes under the root, to a linear tree with the maximum possible height. We also require diversity in the balancedness of the trees. We retain three alternative definitions of balancedness (e.g., see [26]), each yielding low, medium and high cases to cover. Given a height h, we measure *Leaf-balancedness* by the rate of leaves that are either at depth h or $h-1$. The *height-balancedness* metric is the rate of internal nodes that introduce subtrees of similar heights (+/- 1), and similarly for *size-balancedness*. To exclude degenerate covering cases, we only report balancedness for trees having at least 5 nodes, and a height strictly greater than 2.

TaxPayer: We consider three cases for the addresses of a taxpayer: the taxpayer has no address in Luxembourg, at least one address in Luxembourg and another one elsewhere, or all the addresses are in Luxembourg. From constraint C4, we also extract cases combining the various types of local incomes (pension, employment, other) with no address in Luxembourg.

Table III reports the total number of cases to cover for each case study, as well as the coverage results. The complete list of cases to cover is available in our repository [1]. For example, there are 51 cases to cover for Oz. From Table III, 100% of them are covered by each of the 10 test sets. The full coverage is reached after a median number of 16.5 data have been generated (8 in the earliest case, 49 in the latest one). As can be seen, TAF succeeds in supplying a high coverage for all case studies. For Tree, we consider that this was challenging, because the measured properties of the tree structure only indirectly emerge from the instantiation of parameters. There is no notion of balancedness in the model, and still TAF managed to generate trees with diverse balancedness metrics. The only case study for which full coverage is not reached is Bitmap. The missing cases concern high values for the darkest pixel in an image (hence yielding a nearly completely white image) , and similarly low values for the lightest pixel (nearly black image). Such extremal cases are unlikely during the generation process. They are best covered by creating partially instantiated test cases that have the bottom left pixel or the top right one at the desired values.

TABLE III
COVERAGE SUPPLIED BY 100 TEST CASES GENERATED BY TAF

	# cases to cover	% coverage	sample size to reach final coverage
Oz	51 cases	100%	8-49 (median 16.5)
Bitmap	28 cases	89.3%-96.4%	10-80 (median 28)
Tree	18 cases	100%	8-47 (median 22.5)
TaxPayer	70 cases	100%	2-8 (median 2.5)

In comparison, PLEDGE does not provide data diversity. Using PLEDGE, the ten test sets supply a coverage of respectively 52% for Oz and 51% for TaxPayer. The generation manages well a varying number of instances in class architectures, but the data content is the simplest solution returned by the solver. For example, for Oz, the generated fields have a varying number of crop rows, but all rows have an identical length of 10m (the minimum of the interval), the noise in the alignment in crop plants is zero, and so on. For TaxPayer as well, the population of taxpayers lacks diversity, e.g., all taxpayers are born in 1920 (the minimal birth date), their address is never in BE or DE. The authors acknowledge the lack of diversity [17]: they argue that diversity can be effectively enforced by dynamically updating the OCL constraints with new inequalities.

However, this is not as satisfactory as built-in diversity. The user has to very explicitly manage the generation of each numerical parameter, and may still get limited diversity.

For instance, consider how to enforce diverse row lengths for Oz. By default, all rows have the minimal length (10m). We may enforce the first and last rows to be different: the generated lengths are then 10m, ..., 10m, 10.5m. If we further require all rows to be different, the generation time is longer and the different values are still between 10m and 10.5m. We would need additional constraints to cover other values (up to 100m) and other variation patterns between consecutive rows (up to +/-10%). This becomes heavy. The constraints grow more complex than mere inequalities. In contrast, TAF natively provides diverse values and variation patterns.

To conclude, TAF was successful in generating constrained data that provide a high coverage of the data space. To the best of our knowledge, there is no comparable tool that would offer a similar facility based on SMT solving.

VII. CONCLUSION

The paper has presented TAF, a tool that automatically generates test cases from high-level XML-based data models. This tool introduces a novel way to generate diverse instances of data elements in the size-varying data structures. The core feature of TAF is the function "solve_with_diversity", that mixes constraint solving and random sampling. It assigns random values to as many parameters as possible before

solving the constraints, in order to force diversity in the obtained solutions. The function is part of a main algorithm that uses a layered generation approach with backtracking. The top-down decomposition into layers ensures that the data elements are generated only after their number of instances is decided.

TAF has been applied to four case studies. It successfully generated valid and diverse test cases for all of them. The comparison with an alternative tool using the same backend solver showed that the generation time is competitive, while allowing for a much better coverage of the data space.

Regarding improvements of TAF, three main aspects may be envision. First, a connection with de facto standard languages like UML, would improve the usability of our tool. A second point is the native inclusion in TAF of some object-oriented concepts such as inheritance, bidirectional or recursive associations. Particularly, implementing (bounded) recursive associations would allow a more direct expression of some structural properties, which are currently specified by means of quantified predicates rather than by recursive definitions. The third improvement is to interface TAF with a generic language for solver facilities. For instance, an interfacing with SMT-LIB would allow TAF to use several solvers as an alternative to Z3.

Finally, TAF can be used as a test generation tool by itself, but we plan to integrate it in a larger test automation framework. Our future work will consider the integration into a framework that leverages the test completion facility of TAF. We envision an approach where test objectives are produced under the form of partially instantiated test cases, completed by TAF, submitted to the system under test, and then analyzed to produce new objectives. This approach will be experimented in our on-going work on testing autonomous robots in complex 3D environments.

REFERENCES

[1] "Testing Automation Framework," https://www.laas.fr/projects/taf/, 2019, accessed: 2021-11-15.

[2] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary*, 2008, pp. 337–340.

[3] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: Toward a stateful network protocol fuzzer," in *International Conference on Information Security, Samos, Greece*, 2006, pp. 343–358.

[4] https://about.gitlab.com/solutions/dev-sec-ops/, 2021, accessed: 2021-11-09.

[5] D. Marinov and S. Khurshid, "Testera: a novel framework for automated testing of java programs," in *16th Annual International Conference on Automated Software Engineering (ASE), San Diego, USA*, 2001, pp. 22–31.

[6] R. Dutra, J. Bachrach, and K. Sen, "SMTSampler: Efficient stimulus generation from complex SMT constraints," in *International Conference on Computer-Aided Design (ICCAD 2018), San Diego, California, USA*, 2018.

[7] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, vol. 12, no. 3, pp. 366–375, 1972.

[8] A. G. Duncan and J. S. Hutchison, "Using attributed grammars to test designs and implementations," in *5th International Conference on Software Engineering (ICSE), San Diego, California, USA*, 1981, p. 170–178.

[9] P. M. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Software*, vol. 7, no. 4, pp. 50–55, 1990.

[10] D. Coppit and J. Lian, "Yagg: An easy-to-use generator for structured test inputs," in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE), Long Beach, California, USA*, 2005, pp. 356–359.

[11] Z. Xu, L. Zheng, and H. Chen, "A toolkit for generating sentences from context-free grammars," in *8th IEEE International Conference on Software Engineering and Formal Methods (SEFM), Pisa, Italy*, 2010, pp. 118–122.

[12] F. Pan, Y. Hou, Z. Hong, L. Wu, and H. Lai, "Efficient model-based fuzz testing using higher-order attribute grammars," *Journal of Software*, vol. 8, pp. 645–651, 2013.

[13] O. Cekan and Z. Kotasek, "A probabilistic context-free grammar based random test program generation," in *Euromicro Conference on Digital System Design (DSD), Vienna, Austria*, 2017, pp. 356–359.

[14] F. M. Kifetew, R. Tiella, and P. Tonella, "Generating valid grammar-based test inputs by means of genetic programming and annotated grammars," *Empirical Software Engineering*, vol. 22, no. 2, pp. 928–961, 2017.

[15] J. Cantenot, F. Ambert, and F. Bouquet, "Test generation with satisfiability modulo theories solvers in model-based testing," *Software Testing, Verification & Reliability*, vol. 24, no. 7, p. 499–531, 2014.

[16] J. Cabot, R. Clarisò, and D. Riera, "On the verification of uml/ocl class diagrams using constraint programming," *Journal of Systems and Software*, vol. 93, pp. 1–23, 2014.

[17] G. Soltana, M. Sabetzadeh, and L. C. Briand, "Practical constraint solving for generating system test data," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 2, 2020.

[18] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *International Conference on Functional Programming (ICFP), Montreal, Canada*, 2000, pp. 268–279.

[19] R. Feldt and S. Poulding, "Finding test data with specific properties via metaheuristic search," in *IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), Pasadena, CA, USA*, 2013, pp. 350–359.

[20] S. Poulding and R. Feldt, "Generating structured test data with specific properties using nested monte-carlo search," in *Genetic and Evolutionary Computation Conference (GECCO), Vancouver, BC, Canada*, 2014, pp. 1279–1286.

[21] D. Jackson, "Alloy: a language and tool for exploring software designs," *Communications of the ACM*, vol. 62, no. 9, pp. 66–76, 2019.

[22] ——, "Automating first-order relational logic," in *ACM Symposium on the Foundations of Software Engineering (SIGSOFT), San Diego, California, USA*, 2000, pp. 130–139.

[23] "Object Constraint Language," https://www.omg.org/spec/OCL/, 2021, accessed: 2021-11-09.

[24] K. Dewey, L. Nichols, and B. Hardekopf, "Automated data structure generation: Refuting common wisdom," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, pp. 32–43.

[25] C. Robert, T. Sotiropoulos, H. Waeselynck, J. Guiochet, and S. Verhnes, "The virtual lands of oz: testing an agribot in simulation," *Empirical Software Engineering (EMSE)*, vol. 25, no. 3, pp. 2025–2054, 2020.

[26] S.-H. Cha, "On complete and size balanced k-ary tree integer sequences," *International Journal of Applied Mathematics and Informatics*, vol. 6, no. 2, pp. 67–75, 2012.