Pairwise Testing Revisited for Structured Data With Constraints

Luca Vittorio Sartori
LAAS-CNRS, Univ. Toulouse
Toulouse, France
firstname.surname@laas.fr

Hélène Waeselynck
LAAS-CNRS, Univ. Toulouse
Toulouse, France
firstname.surname@laas.fr

Jérémie Guiochet
LAAS-CNRS, Univ. Toulouse
Toulouse, France
firstname.surname@laas.fr

***Abstract*—Pairwise testing (PT) exercises the interactions of pairs of input parameters. The approach is classically defined for a flat set of parameters, the number of which is fixed. Such a definition does not fit well with applications that process structured data like XML and JSON documents. This paper revisits the PT concepts to accommodate hierarchical data structures. The choices and pairs are created by considering the multiplicity of data instances, their access paths and common ancestors. The revised PT approach is implemented on top of on a recent data generation tool, TAF. TAF mixes random sampling and constraint solving to produce diverse data from XML-based models. Our PT implementation interacts with TAF by inserting pair coverage constraints into the models. It monitors overall coverage progress by XPath queries on the data returned by TAF. The approach is demonstrated for two data models: a 3D scene for an agricultural robot, and a population of taxpayers for a tax management system.**

***Index Terms*—pairwise testing, combinatorial testing, software testing, multiplicity, structured data model, test case generation**

I. Introduction

Any software, service, or system, defines an input space that cannot be exhaustively explored. Combinatorial testing [1] is a widespread approach to select test cases in spaces characterized by a set of parameters. The domains of the parameters are partitioned into classes of values, forming the choices for each parameter. Combinatorial testing then covers tuples of choices coming from different parameters, in order to exercise their interaction. Empirical studies have shown that most of the interaction bugs involve the combination of a few parameters. Hence, a widely used combinatorial testing strategy is pairwise testing (PT), where the tuples are simply pairs (checking 2-way interactions).

The PT algorithms classically consider a flat set of parameters, the number of which is fixed. Such a definition does not fit well with applications that process structured data like XML and JSON documents. The input space consists of data elements of various types, embedded into other elements with multiplicity. This cannot easily be brought back to a flat set of interacting parameters. For example, let us assume that an XML input describes a population of person elements, having some age and hobby attributes. The structure and multiplicity cannot be ignored, since the interactions (age, hobby) may be covered in the context of *one* person instance, or of *different* ones. Moreover, there are various population sizes to cover, and various possible shapes of persons (e.g., a person may or not have children). To further complicate things, the shape and numerical content of the data may have to satisfy semantic constraints. For instance, too young persons cannot have children.

Our work revisits the PT concepts to accommodate hierarchical data structures with constraints. The contributions are the following:

- We formalize the PT problem for input data trees labeled by element names. The definitions of choices and pairs consider the multiplicity of elements, their access paths and common ancestors. The coverage checks are expressed by XPath queries on the data trees. To the best of our knowledge, this is the first PT formalization attempt in the context of rich inputs like XML documents.
- We demonstrate an implementation of the formalization on top of a recent tool, TAF [2]. TAF mixes random sampling and constraint solving to generate instances of XML-based data models. We propose two PT algorithms that leverage this tool: 1) unguided, where TAF freely produces data instances and we monitor coverage progress, and 2) guided, where we insert pair coverage constraints into the data models to drive TAF. We also study greedy variants of the algorithms, seeking to cover the greatest number of new pairs at each iteration.
- The algorithms are applied to two examples of data models with constraints: a 3D scene for an agricultural robot, and a population of taxpayers for a tax management system.

Section II discusses related work. Section III presents the formalization of the PT problem, by successively revisiting the concepts of test parameters, choices and pairs. Section IV introduces the implementation on top of TAF, and Section V gives the experimental results for the two case studies. Section VI concludes the paper with future directions

II. Related work

Combinatorial testing (CT) uses a set of parameters to build the test suite, which will contain combinations of choices partitioning these parameters [1]. The CT algorithms generate mathematical objects called covering arrays, which represent test suites at an abstract level. The abstract test cases of the suite, represented by the choices they cover, must then be concretized to come up with real test cases.

There are various techniques to build a covering array [3]: algebraic techniques, greedy algorithms, heuristic search, and constraint satisfaction. Greedy algorithms can work one-row-at-a-time, like AETG [4], or horizontally and vertically expanding the solution, like IPOG [5]. Some CT tools support (propositional) constraints, like ACTS [6] or PICT [7].

The most difficult aspect in the application of CT is the modeling of the input domain, to come up with an adequate abstraction in terms of parameters and choices. One of the popular methods to help in the modeling task is the Classification Tree Method [8] (CTM). CTM provides a structured approach to identify and refine the important characteristics of a system. Note that having a tree model does not mean that the test cases themselves have a tree structure. This is just a convenient abstraction to organize characteristics down to parameters and choices. The tree must be flattened to obtain a set of parameters with their choices, so that CT tools can be used [9]. If the tree has constraints (e.g., a constraint on the presence of related nodes), the flattening algorithm must take care that the node-level constraints are translated into parameter-level ones [10].

Some authors have studied the application of CT for cases where the data is inherently structured, like ontologies or XML documents. Klück et al. propose an algorithm to flatten ontologies [11], so that they can be fed into a CT tool. The flattening manages the inheritance and composition relations, but does not consider other types of dependencies, like the ones introduced by semantic constraints. Multiplicity is managed by having a different variable for each possible instance (hence m variables if the multiplicity is bounded by m), with a special choice epsilon (ϵ) indicating that this instance is absent and has no value. This processing of multiplicity can only accommodate a small m. Borazjani et al. [12] address input models that can have the structure of graphs and apply CT to XML data. Rather than flattening the structure, they propose a compositional approach with unit and integration steps. During integration, the covering arrays from the unit steps are merged to form larger arrays. The compositional building of covering arrays has been further formalized by Kampel et al. [13]. The formalization does not address multiplicity, and the whole approach assumes that the unit models are independent. Constraints relating them could make the CT problem non-compositional.

In contrast to the flattening or compositional approaches, we do not attempt to bring back the test generation problem to a problem (or set of problems) that falls in the scope of CT tools. Rather, we feel free to depart from the classical framework. It allows us to provide a formalization fully tailored to models with structure, a high degree of multiplicity, and constraints relating arbitrary elements. We then demonstrate the feasibility of an implementation. To accommodate constraints, it directly generates concrete test data from the model with the help of a powerful solver (e.g., an SMT solver).

Our formalization establishes a bridge between CT and a large body of work on structured data like XML documents. As a theoretical foundation for such documents, related work has studied tree data models, their specification and queries. The specification uses schema languages, such as DTD, W3C XML Schema, and RELAX NG. They have different expressive powers to define structural shapes [14]. RELAX NG has the full power of regular tree grammars, which define tree languages that can be recognized by tree automata. XSD is less expressive, restricted to a subset called single-type tree grammars, and DTD is even less expressive. In our work, we consider single-type tree models that do not contain recursive definitions. The querying of trees has yielded the development of languages like XPath [15], which we conveniently use for checking the coverage provided by the generated data.

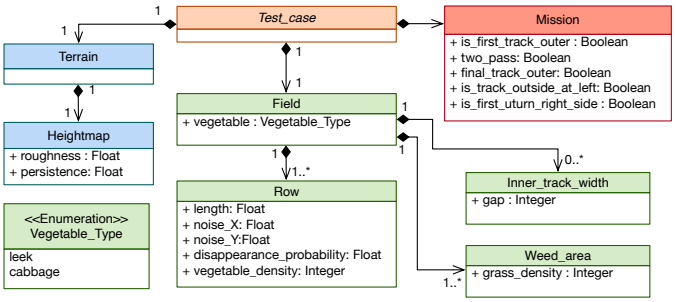

Fig. 1. UML class diagram of the Oz robot case study and running example of the paper.

III. Pairwise Testing with Multiplicity and Structure

This section formalizes pairwise testing from structured data models. It introduces a number of definitions and examples, ended by the symbol $\Diamond$.

A. Data models under consideration

This paper uses the running example of an agricultural robot named Oz [2]. The robot has to perform weeding missions in crop fields. The UML class diagram in Figure 1 specifies test cases for a simulation platform that creates virtual crop fields. A test case is composed of a mission (tuned by a set of parameters), a terrain with its heightmap, and a field containing multiple rows of vegetables. OCL constraints (not shown) further specify the shape and content of a test case.

This example illustrates the kind of test data that we address: hierarchical data structures with multiplicity. The data instances can be modeled as unranked labeled trees, i.e., trees in which nodes can have an arbitrary number of children and are labeled over an alphabet Σ (e.g., $\Sigma = \{\text{“}Field\text{”}, \text{“}Row\text{”}, \text{“}length\text{”}, \dots\}$). Leaf nodes carry a value, e.g., a row length of 15.0 meters. Values are from basic types such as integers, floating point numbers, strings, and Booleans. Let $\mathcal{U}$ be the universe of all such values. We provide below a formal definition for data instance trees (I-trees, for short).

Definition 1 (Data instance tree (I-tree))**.** A data instance over an alphabet Σ and universe $\mathcal{U}$ can be defined as a finite tree structure $T_{\Sigma,\mathcal{U}} = (N, Leaf, Comp, \lambda, Val)$ where:

- N is the set of nodes, also called the tree domain. Classically, we define N as a finite, prefix-closed subset of $\mathbb{N}^*$ (the sequences of natural numbers). Intuitively, the sequences encode the position in the tree. The root node is the empty sequence ϵ, its first child is 0, its second one is 1, and $1 \cdot 2 \cdot 0$ is a grand-grand child. Since the sequences are prefix-closed, if $n \cdot i \in N$ then $n \in N$. We further require that there is no gap in the numbering: if $n \cdot i \in N$, then $n \cdot j \in N$ for all $j < i$.
- $Leaf$ and $Comp$ form a partition of N. $Leaf$ contains all the nodes n such that $n \cdot 0 \notin N$, while $Comp$ contains the other nodes. The root is composite: $\epsilon \in Comp$.
- $\lambda : N \to \Sigma$ is the node labeling function.
- $Val : Leaf \to \mathcal{U}$ is the value function of leaf nodes. $\Diamond$

Query languages for data instance trees (e.g., XPath) refer to the notion of access path, which we introduce below.

Definition 2 (Access path)**.** The access path of a node in a Σ-labeled tree is a sequence of labels recursively defined as follows:

- $path(\epsilon) = \lambda(\epsilon)$, i.e., the access path to the root is its label.
- $path(n \cdot i) = path(n) \cdot \lambda(n \cdot i)$, i.e., the label of a non-root node is appended to the access path of its parent. $\Diamond$

Note that several nodes may have the same access path. In XPath queries, the access paths would be written with the '/' separator, and a query like $/Test\_case/Field/Row$ would return the set of all nodes having this access path. Note also that the definition of I-trees does not distinguish element attributes from atomic elements. They may be distinguished by adopting the convention that attribute labels start by the character '@', like in the XPath query $/Test\_case/Field/Row/@length$.

A data model M like the class diagram in Figure 1 restricts the form of the I-trees which are valid test cases. Examples of invalid I-trees would have zero or multiple field nodes under the root (there must be one), or a row length with the value "Hello World!" (must be a numerical value). The trees must also satisfy OCL constraints, not visible in Figure 1. Likewise, XML schemas (DTD, XSD, Relax-NG) or JSON schemas specify permissible structures and values. They can be further constrained, e.g., by schematron rules based on XPath [16]. All these modeling notations allow for validation against M: given a candidate I-tree, it is decidable whether the tree is a valid instance of M. We note $\mathcal{L}(M)$ the set of I-trees accepted by M. It corresponds to the valid test input domain. If M involves constraints, the existence of any valid I-tree may not be decidable. As a special case, the existence of any valid tree satisfying a test coverage requirement may not be decidable. Hence, our formalization of pairwise testing cannot exclude the production of infeasible pairs. But pair coverage analysis should be possible thanks to validation procedures.

We leave open which modeling notations to use, but assume that the expression of M can be split into two parts: some structural declarations on the one hand, and constraints on the other hand. E.g., M is a hierarchical class diagram plus OCL constraints, or an XML schema plus schematron rules. We note M_{dcl} the model without the constraints. Ignoring the constraints yields more valid trees, hence $\mathcal{L}(M_{dcl}) \supseteq \mathcal{L}(M)$. The structural notation used for M_{dcl} usually has limited expressiveness. For instance, *regular* tree grammars are the common foundation of all XML schema languages [14]. We further require that M_{dcl} possesses the *single-type* tree property: in the context of an element, the subelements with the same label must be declared having the same type. E.g., a crop field may contain multiple row instances, but there is a single Row class defining them all. As a result, the access path of an I-tree node (e.g., $/Test\_case/Field/Row$) suffices to determine an expected type (e.g., the unique Row definition in this context). We finally require that M_{dcl} does not contain recursive definitions: their consideration is delayed to future work. Definition 3 recaps our requirements on the data model.

Definition 3 (Data model over Σ and $\mathcal{D}$)**.** Let Σ be an alphabet, $\mathcal{U}$ a universe of values, and $\mathcal{D}$ a set of value domains in $\mathcal{P}(\mathcal{U})$. Our work considers data models M over Σ and $\mathcal{D}$, composed of a structural declaration part M_{dcl} and a set of constraints C such that:

- M_{dcl} specifies a single-type tree language over Σ.
- M_{dcl} does not contain recursive definitions.
- For each domain $d \in \mathcal{D}$ mentioned in M_{dcl}, it is decidable whether any candidate value $u \in \mathcal{U}$ is in d.
- For each constraint $c \in C$, it is decidable whether any candidate I-tree $T_{\Sigma,\mathcal{U}}$ satisfies c. $\Diamond$

B. Test parameters revisited

In combinatorial testing, test $parameters$ are first identified, then their value domains are partitioned into $choices$, and finally the choices of the various parameters are $combined$. We need to define which parameters should be considered in the data models.

Informally, we will consider all elements declared having a basic type as parameters, i.e., all atomic elements and attributes. For example, a row length is a test parameter. Moreover, we account for the multiplicity of elements: the number of rows is also a parameter. The test coverage will then require I-trees with various sizes and contents.

We extract the parameters from M by processing its structural declaration part M_{dcl}. M_{dcl} may have a design pattern where some reusable definitions are factorized (e.g., inherited classes in UML, global $complexType$ definitions in XSD). The processing has first to expand all the references to reusable definitions, yielding nested element declarations. In this "Russian doll" design pattern, the structure of the model mirrors the one of the data instances, which is convenient for our purposes. We then have a second processing step, which extracts information from the "Russian doll". In the content definition of elements, we only keep the information needed for the identification of test parameters: which subelements may appear under which element, with which possible multiplicity, and with which value domain

(for atomic elements). It yields a test data specification tree (S-tree). An S-tree is very much like an I-tree, but leaf nodes carry value domains rather than values and we add a multiplicity meta-attribute to all nodes. Definition 4 introduces the structure of an S-tree. Definition 5 provides its relation with M_{dcl}, giving correctness requirements for the S-tree extraction.

Definition 4 (Structure of a test data specification (S-tree))**.** An S-tree over alphabet Σ and universe of domains $\mathcal{D}$ is a structure $T_{\Sigma,\mathcal{D}} = (N, Leaf, Comp, \lambda, Type, \mu)$ where:

- N, $Leaf$, $Comp$ and λ are the same as in Definition 1. Note that the nodes of the S-tree have access paths using Definition 2.
- $Type : Leaf \rightarrow \mathcal{D}$ is the function assigning value domains to leaf nodes.
- $\mu : N \rightarrow \mathcal{P}(\mathbb{N})$ is the function assigning a set of multiplicity values to each node. The multiplicity set of the root ϵ must be the singleton $\{1\}$. $\Diamond$

Definition 5 (Correctness of the S-tree wrt M_{dcl})**.** Let M be a data model over Σ and $\mathcal{D}$, and let M_{dcl} be its structural declaration part, defining the tree language $\mathcal{L}(M_{dcl}) \supseteq \mathcal{L}(M)$. The extraction of an S-tree $T_{\Sigma,\mathcal{D}} = (N, Leaf, Comp, \lambda, Type, \mu)$ from M_{dcl} must satisfy the following properties:

- Preservation of the single-type property of M_{dcl} – Two different nodes in $T_{\Sigma,\mathcal{D}}$ cannot have the same access path.
- Access paths are untouched – An access path p exists in $T_{\Sigma,\mathcal{D}}$ if and only if there exists an I-tree $IT \in \mathcal{L}(M_{dcl})$ in which a node has this access path.
- Value domains are untouched – A value v exists, such that $v \in Type(n)$ for some node n of $T_{\Sigma,\mathcal{D}}$, if and only if there exists an $IT \in \mathcal{L}(M_{dcl})$ where v appears under the access path of n.
- Multiplicity is untouched – A value m exists, such that $m \in \mu(n.i)$ for some non-root node $n.i$ of $T_{\Sigma,\mathcal{D}}$, if and only if there exists an $IT \in \mathcal{L}(M_{dcl})$ where a node has the same access path as n and exactly m children labeled $\lambda(n.i)$. $\Diamond$

For our running example, where M_{dcl} is the class diagram shown in Figure 1, the extraction of an S-tree is straightforward. There is no inheritance relation to expand, and the notation directly gives the multiplicity of the composition relations. In other notations like XML schemas, the structural constructs may involve regular expressions. For example, M_{dcl} could define an element containing subelements A, B, C with the following pattern: $A^2B + BC^*$. The S-tree extraction would have to interpret the regex to retrieve the correct multiplicity sets, i.e. $\{0, 2\}$ for A, $\{1\}$ for B and $\mathbb{N}$ for C.

We derive the set of test parameters from the S-tree nodes having a variable multiplicity or carrying values (leaf nodes). Note that the root cannot yield any parameter: it is not a leaf and its multiplicity is always 1. We encode a parameter by a quadruplet. The first three components uniquely identify the parameter, while the fourth one gives its value domain. The identifying triplet consists of 1) a Boolean value indicating the parameter category ($true$ for multiplicity parameter or $false$ for content of a leaf node), 2) its access path split into the parent path (context) and 3) the name of the element of interest. For example, the row multiplicity parameter has the id triplet ($true$, $Test\_case.Field$, Row) and the value domain $\mathbb{N}_1$. This is formalized in Definition 6.

Definition 6 (Set of Test parameters from an S-tree)**.** Let $T_{\Sigma,\mathcal{D}} = (N, Leaf, Comp, \lambda, Type, \mu)$ be an S-tree. The set $PARAM \subseteq \mathbb{B} \times \Sigma^+ \times \Sigma \times \mathcal{D}$ is the one built as follows.

- For each non-root node $n.i \in N$ such that $card(\mu(n.i)) > 1$, create a parameter $(true, path(n), \lambda(n.i), \mu(n.i))$.
- For each leaf node $n.i \in Leaf$, create a parameter $(false, path(n), \lambda(n.i), Type(n.i))$. $\Diamond$

C. Choices revisited

The value domains of parameters are now partitioned into choices. For example, the row multiplicity domain is split into three choices: a small number of rows (say, < 5), a medium one (from 5 to 50) and a large one (> 50). Since we want the coverage of choices to be checkable, we require their definition to explicitly embed a check. The check is a single-variable predicate, written with the usual Boolean, relational and arithmetic operators, and possibly other operators (e.g., on strings). We assume that the choice predicates demonstrably partition the value domain d of the parameter. A choice is then encoded like a parameter, but with a predicate replacing the value domain (e.g., $x < 5$ replaces $\mathbb{N}_1$).

Definition 7 (Set of Choices)**.** Let $PARAM \subseteq \mathbb{B} \times \Sigma^+ \times \Sigma \times \mathcal{D}$ be a set of parameters. Given any parameter $p \in PARAM$, let $PRED_x(p)$ be the set of choice predicates proposed for partitioning its value domain. The set $CHOICE \subseteq \mathbb{B} \times \Sigma^+ \times \Sigma \times \bigcup_p PRED_x(p)$ is the one built as follows:

- for each parameter $p = (b, ctx, label, d)$, for each predicate $pr \in PRED_x(p)$, create the choice $(b, ctx, label, PRED_x(p))$. $\Diamond$

Due to the multiplicity of elements, the coverage of choices may have different meanings. Let us assume that we have a choice for small row lengths (say, $\leq 20.0m$). Is the choice covered when all rows have a small length, or does the existence of one small row suffice? This work adopts the existential interpretation, which seems more natural to us.

How to check whether an I-tree covers a choice depends on the used notation. Many query languages are based on XPath, so we use this notation as an example. Example 1 shows how to build an existential XPath query from a choice.

Example 1 (XPath query for checking choice coverage)**.** Let $c = (b, ctx, label, pr_x)$ be a choice. We assume that the ctx expression has the '/' separator between labels.
If $b = true$ (multiplicity parameter), we produce a string str = "count(" . $label$. ")", where count() is the Xpath function for counting nodes. Otherwise $str = label$.
Then, we substitute str for x in pr_x: $pr = pr_x[str/x]$.
And finally we build the query:

"count(" . ctx . "[" . pr. "]) > 0".
For the choice of a small number of rows, this yields:
starting from $c = (true, Test\_case/Field, Row, x < 5)$, the string str = "count(Row)" is produced,
then pr = "count(Row) < 5", and finally
"count(/Test_case/Field[count(Row) < 5]) > 0".
This query searches for the set of field nodes that have a small number of row children, and then checks that the node set is not empty. Similarly, the choice of a small row length gives: "count(/Test_case/Field/Row[@length ≤ 20.0]) > 0" ◊

D. Pairs revisited

This paper focuses on pairwise testing, i.e. on the coverage of pairs of choices. For example, a pair requires both a small row length and a high roughness of the terrain, or a small row length combined with a small number of rows. In the usual applications of pairwise testing, the pairs combine all possible choices from any two different test parameters. Then, given an input data, the coverage of the pair (c_1, c_2) conjoins two independent checks: for the coverage of c_1 and the one of c_2.

The creation and checking of pairs are more complicated in our case, because the parameters are embedded into a data structure with multiplicity. First, the existence of self pairs becomes possible, which come from the *same* parameter. E.g., multiple rows are possible, hence multiple row lengths occur and we may create pairs testing their interaction. Second, it is less clear how to check for pair coverage. Assume we have a pair (small row $length$, low $vegetable\_density$). Do we mean that the same row has to cover both choices, or can each choice occur independently? The question can be reformulated in terms of the lowest (i.e., farthest from the root) common ancestor of the two covering nodes. In an I-tree, a length node and a vegetable density node can have two types of lowest common ancestor (LCA): $Field$ or Row. If we admit any of them for the coverage of the pair, the checks of the choices can be independent. If we admit only Row, then the checks are no longer independent: we must search for a row that covers both choices at the same time. Another example is the self pair of two small row lengths. The LCA must be a $Field$ (since a row has a single length). Again, the checks cannot be independent: we need a single check, asking for at least two small rows in a field.

From what precedes, self pairs and LCAs are new aspects that justify revisiting the notion of pairs. We provide below a generic formalization of pairwise strategies in terms of two meta-parameters: $Self$, determining whether or not to create self pairs, and $Select()$ determining how to select the permitted LCAs of a pair. We will later give examples of LCA selection functions and XPath coverage checks.

The formalization first introduces the notion of *divergence point* in the context of a choice (or more generally a parameter), where the context is the access path of its parent (cf. Definition 7). Divergence points are where multiplicity may cause several branches in the I-tree. For example, a row length has the context $Test\_case/Field/Row$, and $Test\_case/Field$ is a divergence point. Under a field node, there may be branches for the first row instance, the second one, and so on. A row multiplicity parameter also has the context $Test\_case/Field/Row$, but there is no divergence point. Indeed, there is a single number of rows in the field. Definition 8 formalizes this. Note how the recursive definition of the set of divergence points ($sdiv$, updated as $sdiv'$) considers multiplicity parameters (Boolean b is true) like parameters that have at most one instance.

Definition 8 (Divergence point)**.** Let c be a choice $(b_c, ctx_c, label_c, pred_c)$ created from an S-tree $T = (N, Leaf, Comp, \lambda, Type, \mu)$.
Let $node : \Sigma^+ \to N$ be the function returning the node of T having a given access path. The functions exists since access paths are unique in the S-tree (Definition 5).
Let $div : \mathcal{B} \times \Sigma^+ \times \Sigma \times \mathcal{P}(\Sigma^+) \to \mathcal{P}(\Sigma^+)$ be the recursive function defined as follows: $div(b, ctx, label, sdiv)$ =

$$\begin{cases} sdiv' \text{ if } node(ctx) \text{ is the root node } \epsilon, \\ div(false, ctx', label', sdiv') \text{ otherwise,} \end{cases}$$

Where:

- $sdiv' = sdiv$ if $b = true$ or $\mu(node(ctx.label) \subseteq \{0, 1\}$,
- $sdiv' = sdiv \cup \{ctx\}$ otherwise,
- $ctx' \in \Sigma^+$ and $label' \in \Sigma$ are such that $ctx = ctx'.label'$.

The set of divergence points of c is $DIV(c) = div(b_c, ctx_c, label_c, \emptyset)$. Since it contains prefixes of ctx_c, the "is-a-prefix-of" string relation, noted $\prec$, is a strict total order on $DIV(c)$. ◊

Definition 9 uses divergence points to determine the set of possible LCAs for a pair of choices. This is the set from which $Select()$ will extract a subset of allowed LCAs. Basically, the longest common prefix of the contexts of the choices is a possible LCA, and so are all common divergence points. For example, the row length and row vegetable density can be under the same row element (prefix), or under the same field (divergence point) but not the same row. We must however take care of the special case of self pairs. For them, it would be meaningless to consider the common prefix if this is not a divergence point. For example, there is nothing such as a common row parent for two row length choices: each row has a single length. Definition 9 explicitly handles this.

Definition 9 (Set of possible LCAs for a pair of choices)**.** Let $c_1 = (b_1, ctx_1, label_1, pred_1)$ and $c_2 = (b_2, ctx_2, label_2, pred_2)$ be two choices. Let $isSelf(c_1, c_2)$ be the predicate: $b_1 = b_2 \wedge ctx_1 = ctx_2 \wedge label_1 = label_2$. Let $prefix$ be the longest common prefix of ctx_1 and ctx_2. The set of possible LCAs for the pair is the following one: $LCAS(c_1, c_2)$ =

$$\begin{cases} DIV(c_1) \text{ if } isSelf(c_1, c_2), \\ (DIV(c_1) \cap DIV(c_2)) \cup \{prefix\} \text{ otherwise.} \end{cases}$$

Like the sets of divergence points, the sets of possible LCAs are totally ordered by $\prec$. Each non-empty set has a well-defined least element, and a greatest one. ◊

We are now equipped to define a pairwise strategy in a very generic way. It creates pairs of choices, which may contain self pairs or not, and associates a set of permissible LCAs to them. In Definition 10, we encode this as the production of quadruplets of the form (ch1, ch2, set of possible LCAs, set of permitted LCAs). In this way, the encoding not only makes it explicit which LCAs are desired, but also which LCAs do not count for coverage (retrieved by making the set difference).

Definition 10 (Pairwise strategy with meta-parameters $Self$ and $Select()$)**.** Let $Self \in \mathbb{B}$ be the Boolean parameter of the strategy, where a value $true$ indicates that self pairs are to be created, and $false$ that they are not.
Let $Select : \mathcal{P}(\Sigma^+) \rightarrow \mathcal{P}(\Sigma^+)$ be the selection function of the strategy. Its input must be a non-empty, totally-ordered set of access paths, from which it extracts a non-empty subset.
Let $CHOICE$ be the set of choices according to Definition 7. The strategy creates a set $PAIRS$, where each element of the set is in $CHOICE \times CHOICE \times \mathcal{P}(\Sigma^+) \times \mathcal{P}(\Sigma^+)$. The set is the one obtained from the following procedure:

- For each $c_1 \in CHOICE$, for each $c_2 \in CHOICE$ such that $\neg isSelf(c_1, c_2)$, create the pair $(c_1, c_2, LCAS(c_1, c_2), Select(LCAS(c_1, c_2)))$.
- If $Self = true$, for each $c \in CHOICE$ such that $LCAS(c, c) \neq \emptyset$, create the pair ($c$, c, $LCAS(c, c)$, $Select(LCAS(c, c))$). ◊

Let us now discuss examples of $Select()$ functions:

- Don't care – all possible LCAs are permitted. $Select()$ is the identity function.
- Closest-possible – $Select(S)$ returns the greatest element of S. E.g., a row length and vegetable density must be covered in the context of a same row. Intuitively, the strategy considers that the closer the nodes are in an I-tree, the stronger their semantic interaction is.
- Close-enough – The strategy relaxes the closest possible constraint, but still considers a boundary LCA for closeness acceptance. For example, the possible LCAs are $lca_1 \prec lca_2 \prec lca_3$, and the strategy only retains the LCAs starting from lca_2. For a boundary lca_b, $Select(S)$ returns $\{lca \in S \mid lca_b \preceq lca\}$. Intuitively, the strategy is useful when constraints make the closest-possible strategy infeasible for some pairs.
- Arbitrary LCA – The strategy selects one arbitrary LCA for each pair of choices. E.g., for some reason, we want the small row length and high vegetable density to occur in *different* rows, hence the LCA must be a field.

Depending on the strategy, the coverage checks can become quite complex. Examples 2 to 4 illustrate cases of increasing difficulty using XPath queries.

Example 2 (XPath queries for checking coverage when no LCA is removed)**.** We discuss cases where $Select()$ retains the entire set of possible LCAs, either because it is the don't care strategy or because the set is a singleton.
For a non-self pair, each choice can be independently checked, where the XPaths queries are like in Example 1.
For self pairs such that the choices are actually different (same parameter, but different choice predicates to check), two independent checks still do the job. But for self-pairs with the same predicate (e.g., a small row length in both cases), it would not work to repeat the same existential check twice. Rather, we count the number of occurrences of the choice, and checks whether it is > 1. ◊

Example 3 (XPath queries when distant LCAs are removed)**.** The closest-possible and close-enough strategies only remove LCAs that are lower (according to $\prec$) than a boundary LCA. The boundary may be the greatest LCA (closest possible strategy), or another LCA. We consider pairs for which at least one lower LCA is removed (otherwise, see Example 2). Let ctx_1 and ctx_2, be the contexts of the choices of the pair, and l_1 and l_2 their labels. The boundary LCA is a prefix of both contexts, so we have: $ctx_1 = lca_b.suf_1$ and $ctx_2 = lca_b.suf_2$, where the suffixes can be empty.
The individual queries for the choice coverage have the general form: count (/ ctx [$predicate(label)$]) > 0.
We can manipulate the choice queries to get a single check: count (/ lca_b [$pred_1(suf_1/l_1)$] [$pred_2(suf_2/l_2)$]) > 0.
This check requires the coverage of the pair in the context of lca_b and its descendants. ◊

Example 4 (XPath queries for an arbitrary LCA)**.** We discuss the case where the selected LCA is not the greatest one (otherwise, see Example 3). The XPath checks become convoluted. We provide hints on how to do, with the help of an example: a pair (small row length, low vegetable density). The possible LCAs are:
$Test\_case/Field \prec Test\_case/Field/Row$.
The target LCA is the field. The key is to split the common context according to the first node after the target (here, Row). We would like to find at least two *different* rows, such as one has a small length and the other has a low vegetable density. Unfortunately, XPath is not convenient for this.
We may proceed by three checks performed in parallel. If any one succeeds, then the pair is covered. Each check may involve several queries. The three checks are:

1) Count the number of rows that cover both choices. If the number is > 1, we're done.
2) Search for a row that covers the first choice but not the second one. Search for a row that covers the second choice. If both searches succeeds, we're done.
3) Same as 2), by permuting the first and second choices. ◊

E. Adding constraints

The construction of the pairs is from an S-tree, which ignores semantic constraints and even some structural ones (e.g., the regular expressions in the content models of XML schemas). Some of these constraints may involve disjunctive cases, like a content model having to match the pattern A^2B+BC^*, or a logical constraint of the form $pred_1 \vee pred_2$. We leave open whether such cases should generate new parameters and choices, like predicate parameters with truth

value choices. The risk is to get many pairs that are either redundant with existing pairs or infeasible. The improvement in the coverage may not be worth the combinatorial growth. In the case studies we performed, we opted for a compromise: we automatically generate predicate parameters and choices from the constraints, but do not combine them with the others. They yield single choice coverage requirements. Other strategies could have been possible.

Once the list of pairs and single choices is ready, we need a means to produce the covering test cases. The next sections present a solution using a generation tool that mixes random sampling and constraint solving. It allows us to demonstrate an implementation of our (so far, theoretical) PT framework.

IV. Integration with a test case generator (TAF)

TAF (Test Automation Framework) [2] is a recent tool for generating data structures with constraints. It is open-source [17]. The data models must be written in a tool-specific language based on XML. We briefly present this language that fits well into our formalization assumptions. Next, we describe how to implement pairwise testing on top of TAF.

A. TAF input models

The TAF inputs models are called templates. Listing 1 gives a snippet of the template for Oz, our running example. It contains four types of XML elements: Root, Node (for composite elements, like a row), Parameter (for attributes, like a row length), and Constraint. The parameters can be of type Boolean, string, real, and integer. All value domains must be bounded. Numerical parameters (real and integer) are declared with a min-max range of values. For instance, the row length (Line 6) is between 10.0 and 100.0m. Strings are actually enumerated types, like the set {“cabbage”, “leeks”} for the vegetable parameter (Line 3). The parameters and nodes have a nb_instances (number of instances) meta-attribute for multiplicity. It has the integer type. A field element (Line 2) has a single instance. The multiplicity of rows (Line 4) is in the range min=1, max=100.

If one ignores constraints, the structure of declarations is simple. It satisfies the single-type tree property. It specifies bounded data instance trees, both horizontally (bounded multiplicity) and vertically (no recursive definition). The language only allows for a Russian doll design pattern (nested declarations, no reusable definitions). The extraction of an S-Tree from a TAF template is direct.

Constraints add expressiveness. For example, the constraint in Line 16 relates the number of rows to a mission parameter. The logical operators have a function notation, e.g., IMPLIES (-,-). This is to facilitate the communication with the backend solver, Z3 [18]. The navigation operator is ‘\’, as in the path expression “ ..\field\row.nb_instances”. Constraints may have universal or existential quantifiers. In Lines 10-12, the quantified variable i allows the expression of a universal property of rows: consecutive rows must have the same length +/-10%. Other constraints not shown in the snippet concern the content pattern of a field (for n rows, it must have $n+1$ weed areas and $n-1$ inner widths).

TAF outputs an XML file for each generated data instance. It gives us an I-tree, to which we apply XPath coverage queries.

```
1  <root name="test_case">
2      <node name="field" nb_instances="1">
3          <parameter name="vegetable" type="string" values=
       "cabbage;leek"/>
4          <node name="row" min="1" max="100">
5             <parameter name="length" type="real"
6                min="10.0" max="100.0"/>
7                < .../>
8          </node>
9          <.../>
10         <constraint name="interval" types="forall"
11             expressions="row[i]\length INFEQ 1.1*row[i
       -1]\length;row[i]\length SUPEQ 0.9*row[i-1]\length"
12             quantifiers="i" ranges="[1, row.nb_instances
       -1]"/>
13
14     <node name="mission" nb_instances="1">
15         <parameter name="is_first_track_outer" type="
       boolean"/>
16         <constraint name="first_track" expressions="
       IMPLIES(..\field\row.nb_instances EQ 1, .\
       is_first_track_outer EQ True)"/>
17     </node>
18     < .../>
19 </root>
```

Listing 1. Snippet of the TAF XML template for the robot Oz

B. Implementation of pairwise testing on top of TAF

We have developed a parser of TAF templates, which extracts an S-tree and calculates the choices and pairs. For demonstration purposes, the choices are created by a systematic procedure. The definition range of a numerical parameter is uniformly split into three subranges, low, $medium$, and $high$. For a number of instances, we consider specifically the value zero if it is in the range, and split the rest of the domain into low, $medium$, and $high$ values. Enumerated types yield one choice per defined value. The pairs are then formed according to the closest-possible LCA selection strategy with self-pairs.

We also add single choices extracted from the constraints. The constraint expressions are parsed and put in a disjunctive normal form. Then, each expression $OR(cond_1, ..., cond_n)$ yield n single choices such that $cond_i$ $(i = 1, .., n)$ is true. We do not try to combine the truth values of the conditions (e.g., $cond_1$ is true and the others false), nor do we combine these choices with the parameter choices to form pairs. If the constraint has quantifiers, we interpret all of them as if they were existential. We produce an existential Xpath check for each choice.

We consider several algorithms for the generation of covering test cases.

The first one is *unguided*. Since TAF has a randomized generation procedure, it produces different valid cases each time it is called. Then, the simplest algorithm is to ask TAF to produce a test case, apply coverage checks, and retain the test case if it covers new elements (new pairs or single choices). The procedure is repeated until all elements are covered or a maximal number of iterations is reached.

```
<constraint name="sentence_286" types="exist;exist"
 expressions="AND(.\field[a]\row[b]\length SUPEQ 10.0, .\
     field[a]\row[b]\length INFEQ 40.0, .\field[a]\row[b]\
     vegetable_density SUPEQ 0, .\field[a]\row[b]\
     vegetable_density INFEQ 3)"
 quantifiers="a;b"
 ranges="[0, .\field.nb_instances-1]; [0, .\field[a]\row.
     nb_instances-1]" />
```

Listing 2. A coverage constraint added under the root of the Oz template. Note how the path prefixes with common indexes a, b enforce the desired LCA (here, a row).

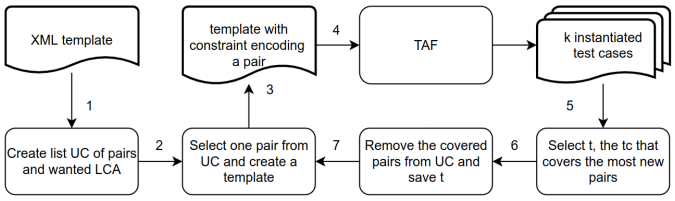

Fig. 2. Flowchart of the guided generation with k trials.

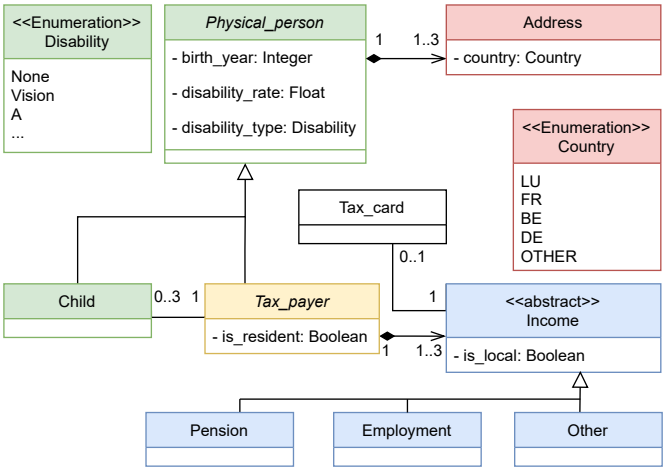

Fig. 3. UML class diagram of the Taxpayer management application case study.

The second one is *guided*. Since TAF handles constraints, we can insert covering constraints into the template. For instance, Listing 2 shows the constraint for the pair (small row length, small vegetable density), automatically translated into the TAF language. The iterative building of a test suite is then as follows: select a pair (or single choice) remaining to cover, insert its coverage constraint into the original template and ask TAF to return a test case. If TAF fails to return one, the pair is put into a list of likely infeasible pairs, made available for manual inspection. If it returns one, then the overall coverage is monitored and all newly covered elements are removed from the To-do list. The procedure is iterated until the list is empty. The resulting test suite covers all the elements that TAF could satisfy.

Both the guided and unguided generation can be extended by a *greedy* procedure that tries to minimize the size of the test suite. It consists in requesting k test cases at each iteration, and retaining the one that covers the greatest number of new elements. This optimization is loosely inspired by the AETG algorithm [4], which also tries different test cases. In the unguided generation, the k cases are freely produced by TAF, with the hope that many new pairs will occur in at least one of them. In the guided generation, the k test cases all cover a target pair, plus hopefully many other ones. The guided-greedy procedure is shown in Figure 2. Note that $k = 1$ corresponds to the original algorithm without optimization.

V. Case Studies

To demonstrate the approach, two case studies are used. Both have their template available in the TAF repository [17]. The first one, Oz, is the running example of this paper. It comes from an industrial case study, where an agricultural robot was tested in simulation [19]. The second one, Taxpayer, describes the data for an income tax management application [20]. The UML class diagram of Oz can be seen in Figure 1, and the one of Taxpayer in Figure 3. The Oz template has 7 composite elements, 15 parameters, and 5 constraints. TaxPayer has 9 composite elements, 12 parameters and 7 constraints. Since TAF only supports composition relations, the inheritance is flattened in the template, and associations are modeled by unidirectional compositions.

A. Experiments applying the PT algorithms

Table I shows the number of pairs and single choices automatically created from each template. A preliminary run of the guided generation algorithm allowed us to identify the infeasible pairs, which were confirmed by a manual analysis.

For each case study, we performed experiments to compare the coverage and test suite size supplied by the various generation algorithms. The code is available in a replication package [21].

The unguided generation allows TAF a max number of 100 iterations to get a covering suite. It is combined with a greedy optimization with k trials, where k = 1 (TAF baseline), 3, 5, 7. The guided approach, inserting covering constraints into the templates, is also combined with a greedy optimization. In total there are 16 experiments: 8 for each case study—4 unguided and 4 guided—, with "k" of values 1, 3, 5, 7. Each experiment was repeated 50 times (so 50 runs). In each run, the timeout given to TAF to generate a valid test case was 120s in the case of Oz and 200s for Taxpayer, decided empirically. To keep the overall time of the 50 runs tractable, we filtered out the infeasible pairs. Hence, the unsatisfied pairs at the end of a run are all satisfiable.

Tables II and III show the results we obtained. The first four columns, labeled TAF, correspond to the unguided approach where TAF freely produces test cases. The next four correspond to the guided generation. The tables display the supplied coverage, the mean generation time of a run, the mean number of unsatisfied pairs (if any) and the mean size of the produced test suite (i.e., the number of tests cases it contains).

The next sections analyze the coverage and test suite size in greater detail.

TABLE I
PAIRS AND SINGLE CHOICES CREATED FROM THE TEMPLATES

	# Pairs	# Feasible pairs	# Single choices
Oz	1328	1245	2
TaxPayer	2016	1946	13

TABLE II
OZ: MEAN COVERAGE PERCENTAGE (COV %), GENERATION TIME (GEN T) IN SECONDS, NUMBER OF UNSATISFIED PAIRS, AND NUMBER OF TEST CASES IN THE RETURNED TEST SUITE.

Oz	**TAF k=1**	**TAF k=3**	**TAF k=5**	**TAF k=7**	**k=1**	**k=3**	**k=5**	**k=7**
Cov%	97.3	97.8	98.4	98.5	100	100	100	100
gen t (s)	171.8	152.8	164.8	184.8	39.2	39.5	41.3	44.2
unsat pairs	33.1	27.0	20.1	18.9	0.0	0.0	0.0	0.0
# tc	21.5	18.4	17.6	17.0	22.6	18.7	17.6	17.0

TABLE III
TAXPAYER: MEAN COVERAGE PERCENTAGE (COV %), GENERATION TIME (GEN T), AND NUMBER OF TEST CASES IN THE RETURNED TEST SUITE. THE NUMBER OF UNSATISFIED PAIRS IS OMITTED, BECAUSE THE VALUE IS ALWAYS ZERO

Tax-payer	**TAF k=1**	**TAF k=3**	**TAF k=5**	**TAF k=7**	**k=1**	**k=3**	**k=5**	**k=7**
Cov%	100	100	100	100	100	100	100	100
gen t (s)	42.2	49.4	61.5	62.6	27.6	40.7	51.3	54.2
# tc	9.3	7.2	6.8	6.2	7.9	5.8	5.4	5.3

B. Coverage

A first observation is that TAF produces sufficiently diverse data for all PT algorithms to be effective. The coverage rate is nearly 100% in each case. However, for Oz, the unguided approaches with k=1,3,5,7 has a mean number of unsatisfied pairs equal to 33.1, 27.0, 20.1, 18.9, compared to the guided approaches having none. It justifies the use of the guided approach to ensure coverage.

For Taxpayer, all the approaches quickly achieve 100% coverage. For Oz, it is interesting to analyze the evolution of the coverage at each iteration of the generation. Figure 4 displays the cumulative coverage at each iteration. The shown experiments are the guided approach with k=1 and k=7, and the unguided approach with k=1 (TAF baseline). The guided approach with k=7 has the steepest growth in coverage and reaches 100% coverage with fewer iterations than the other approaches. The guided k=1 approach also arrives at 100% coverage, but needs more iterations. The unguided approach has the slowest growth. It also has the highest variability, as shown by the shaded areas in Figure 4 that visualize the interquartile ranges.

C. Test suite size

Minimizing the test suite size is important if the tests have to run every time new code is pushed.

The effect of k in the reduction of the test suite size can be seen in Figures 5 and 6, where the first four boxplots are

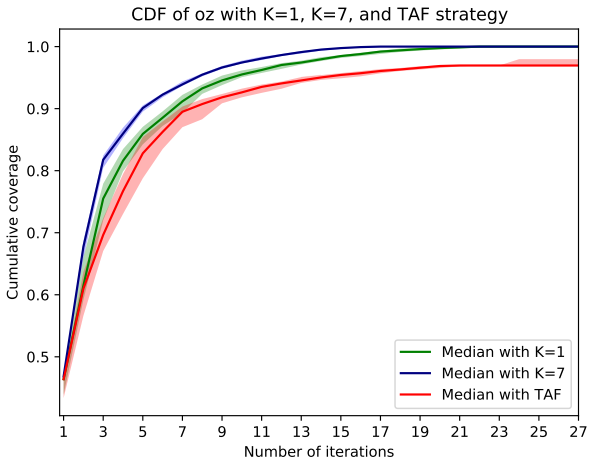

Fig. 4. Oz: comparison of the cumulative coverage with the unguided TAF approach with k=1 (TAF baseline) and the guided approach with k=1 and k=7. The shaded areas represent the interquartile range. The y-axis is limited to the upper part of the diagram for better readability. The guided approach median lines are prolonged to make the comparison visually easier. We do not show k=3 and k=5 for the guided approach, because their diagrams would fit between the diagrams of k=1 and k=7.

for the unguided approach (TAF) with k=1,3,5,7, and the four next for the guided one. Note that it may seem that, for Oz, the unguided approach found the minimum of all sizes (14). However, let us recall that the coverage is not 100% in that case. Actually, the minimal size of all found covering suites is 15.

Inside each family of approaches (unguided or guided), the higher the k, the lower the median size tends to be, and also the lower the minimal size found over 50 runs, and the lower the maximal size. The shift toward smaller values also surfaces by looking at mean values. From Tables II and III, the tendency seems clear: inside each family of approaches, the mean test suite size always decreases when k increases.

We applied a Mann-Whitney U test to determine the statistical significance of the improvement in the mean test size. The first four rows of Table IV shows the p-values when comparing two guided approaches with different values of k . E.g., the first row compares the mean test sizes supplied by the guided approaches for k=1 and 3. The last two rows compares guided and unguided approaches for the same k. For example, "k=1, TAF1" means the guided approach with k=1 versus the unguided one (TAF) with k=1. The comparison is done for TaxPayer only, because it would not make sense to compare test sizes for different coverage results. In each row, we also provide the Common-Language Effect Sizes (CLES).

The p-values are almost always inferior to 0.05, meaning that it is highly unlikely that the effect is caused by chance. There is just one case in which the p-value is greater than 0.05: Taxpayer when comparing k=5 and k=7 (p-value=0.12). The Common-Language Effect Sizes (CLES) show that there are no small effect sizes (>0.2), there are some medium effect sizes (>0.5), and the others are all large effect sizes (>0.8).

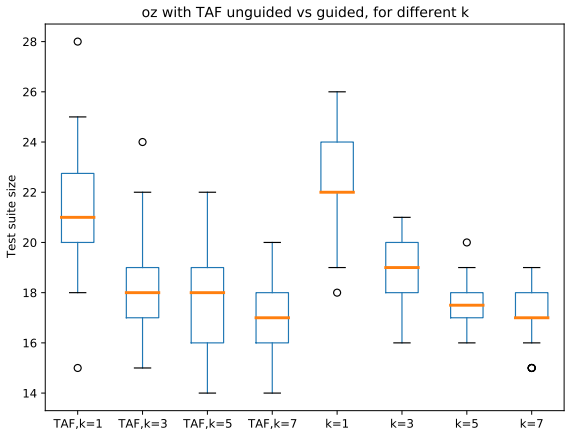

Fig. 5. Oz: Boxplots with TAF unguided with k=1,3,5,7 and guided with k=1,3,5,7.

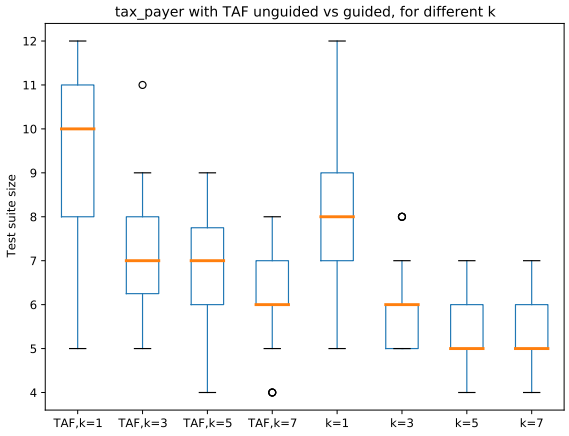

Fig. 6. TaxPayer: Boxplots with TAF unguided with k=1,3,5,7 and guided with k=1,3,5,7.

The lowest effect size (CLES=0.5596) is related to Taxpayer when comparing k=5 and k=7, which has also the highest p-value. The first row of the table shows that, for the guided approach, the move from the no-optimization case (k=1) to a limited optimization (with only k=3 trials) already has a significant beneficial impact.

We conclude that, for the case studies, the guided generation with few optimization trials is a practical approach to produce covering test suites of small sizes.

VI. Conclusion

This paper has presented a formalization of pairwise testing in the case of structured data models with constraints and multiplicity. We identified two major differences with the usual "flat" case. First, self pairs become possible, where the parameter of an element occurs in different instances of this element, and we want to test the interactions between these instances. Second, the tested pairs will not be the same depending on the lowest common ancestor (LCA) of the covering element instances. We proposed several strategies to select the LCAs that will count for coverage. We also provided an XPath formalization of the pair coverage analysis.

TABLE IV
Statistical analysis of the mean test suite sizes. Notation k=i means guided, with i equal to the k under analysis. TAFi means unguided with k=i.

	Oz		Taxpayer	
	P-value	**CLES**	**P-value**	**CLES**
k=1, k=3	6.98E-16	0.96	3.93E-11	0.8674
k=3, k=5	0.000005	0.7488	0.010377	0.6212
k=5, k=7	0.001993	0.6564	0.119922	0.5596
k=1, k=7	2.45E-18	0.9966	3.89E-15	0.9392
k=1, TAF1	N/A	N/A	0.000045	0.7246
k=7, TAF7	N/A	N/A	2.86E-17	0.9762

As a proof-of-concept demonstration, we implemented pairwise testing on top of a data generation tool, TAF. TAF has an input language based on XML and accommodates constraints. We automatically created the pairs to cover and their coverage checks, according to our formalization. We then proposed several generation strategies to build covering test suites with the help of TAF. We applied them to two case studies from real world applications, and successfully produced test suites with 100% coverage of the feasible pairs. A practical approach is to insert the covering constraints into the data models. This guided approach helps to cover pairs that are hard to cover with the native randomized generation of TAF. Moreover, the size of the resulting test suite can be significantly reduced by combining the guided approach with a simple greedy optimization: a few candidate test cases are generated to cover a pair, from which we retain the one providing the highest overall coverage progress.

Future directions include conceptual extensions to the framework as well as alternative implementations. An extension would be to account for recursive definitions, with new parameters related to the depth of the recursion, and new self pairs between elements at a different depth. The generalization to n-way testing may also be considered, but would most probably be limited to *don't care* LCA selection strategies. Otherwise, the LCA analysis and coverage checks could become cumbersome. Alternative implementations of the generation would add techniques that have been found useful in classical PT, like optimizing the order of the pairs (for the guided approach) or starting from pre-existing seed tests.

Acknowledgment

This project has received funding from the European Union's EU Framework Programme for Research and Innovation Horizon 2020 under Grant Agreement No. 812.788.

References

[1] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*, 1st ed. Chapman & Hall/CRC, 2013.

[2] C. Robert, J. Guiochet, H. Waeselynck, and L. V. Sartori, "TAF: a tool for diverse and constrained test case generation," in *21st IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Hanan Island, China, Dec. 2021.

[3] D. R. Kuhn, R. Bryce, F. Duan, L. S. Ghandehari, Y. Lei, and R. N. Kacker, "Chapter one - combinatorial testing: Theory and practice," in *Advances in Computers*, A. Memon, Ed. Elsevier, 2015, vol. 99, pp. 1–66.

[4] D. Cohen, S. Dalal, A. Kajla, and G. Patton, "The automatic efficient test generator (AETG) system," in *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*, 1994, pp. 303–309.

[5] 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)*, 2007, pp. 549–556.

[6] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "ACTS: A combinatorial test generation tool," in *Verification and Validation 2013 IEEE Sixth International Conference on Software Testing*, 2013, pp. 370–375, ISSN: 2159-4848.

[7] "Pict microsoft github," accessed 2022-10-21. [Online]. Available: https://github.com/microsoft/pict

[8] M. Grochtmann and K. Grimm, "Classification trees for partition testing," *Software Testing, Verification and Reliability*, vol. 3, no. 2, pp. 63–82, 1993.

[9] E. V. Sandin and R. Mohamad, "Enhanced classification tree method for modeling pairwise testing," *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, vol. 9, no. 3, pp. 87–92, 2017, number: 3-4.

[10] T. Kitamura, A. Yamada, G. Hatayama, C. Artho, E.-H. Choi, N. T. B. Do, Y. Oiwa, and S. Sakuragi, "Combinatorial testing for tree-structured test models with constraints," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 141–150.

[11] F. Klueck, Y. Li, M. Nica, J. Tao, and F. Wotawa, "Using ontologies for test suites generation for automated and autonomous driving functions," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018, pp. 118–123.

[12] M. N. Borazjany, L. S. Ghandehari, Y. Lei, R. Kacker, and R. Kuhn, "An input space modeling methodology for combinatorial testing," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 372–381.

[13] L. Kampel, B. Garn, and D. E. Simos, "Combinatorial methods for modelling composed software systems," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 229–238.

[14] M. Murata, D. Lee, M. Mani, and K. Kawaguchi, "Taxonomy of XML schema languages using formal language theory," *ACM Transactions on Internet Technology*, vol. 5, no. 4, pp. 660–704, Nov. 2005.

[15] "XML Path Language (XPath)," accessed 2022-10-21. [Online]. Available: https://www.w3.org/TR/1999/REC-xpath-19991116/

[16] E. Siegel, *Schematron: A language for validating XML*. XML Press, 2022.

[17] "Testing Automation Framework," https://www.laas.fr/projects/taf/, 2019, accessed 2022-10-21.

[18] 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.

[19] 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.

[20] 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, pp. 1–48, Apr. 2020.

[21] "Replication package for pairwise testing revisited for structured data with constraints, ICST 2023," accessed 2023-01-23. [Online]. Available: https://redmine.laas.fr/projects/taf/repository/taf-pairwise