Combinatorial Interaction Testing for Test Selection in Grammar-Based Testing

Elke Salecker, Sabine Glesner
Berlin Institute of Technology
TU Berlin – Germany
{*elke.salecker*|*sabine.glesner*}@*tu-berlin.de*

***Abstract*—Systematically enumerating derivations of a grammar yields for realistic grammars test sets that are to large to be tested with reasonable costs. Existing reduction techniques for grammar-based testing guide the enumeration process to restrict the number of generated test cases. However, they do not provide a rule coverage criterion, i.e., they do not aim at providing a test set that ensures coverage of t-wise rule combinations. Selecting derivations such that all possible t-wise rule combinations are covered with at least one test case allows for the generation of small test sets. We employ combinatorial interaction testing to generate the test sets and derive the necessary test specification from the grammar specification automatically. Our evaluation results are twofold. First, they demonstrate the efficiency of our approach. Second, they reveal shortcomings of existing tools for combinatorial interaction testing with constraints.**

***Keywords*-grammar-based testing; rule coverage criterion; combinatorial interaction testing; test specification generation**

I. Introduction

Grammar-based testing allows for the automatic generation of test sets from a context-free grammar. It has been widely used to test parsers and compilers. Moreover, since it is not restricted to this type of software systems, it has been used to test Java virtual machines [1] or serialization frameworks [2]. Test sets are derived by systematically enumerating derivations of growing length. The generated test sets grow exponentially and for realistic grammars already small length values lead to test sets that are too large to be tested exhaustively. Consequently, techniques to select test cases are required. This problem has been addressed with approaches that control the generation process in order to restrict the generated test set. None of these strategies is dedicated to ensure coverage criteria with respect to the use of the grammar rules. The test selection techniques presented in the literature do not aim at ensuring that each possible t-wise rule combination is covered with at least one test case.

In this paper, we present a novel approach for grammar-based test set generation that ensures a required rule coverage criterion. Our approach is a combination of grammar-based testing and combinatorial interaction testing (CIT). CIT is a specification-based technique for the automatic generation of test sets. A CIT generation algorithm calculates a test set for a given specification. It ensures interaction coverage for all value combinations and aims at generating as few as possible test cases.

We observed that the finite set of all derivations with length k can be automatically mapped onto a CIT test specification. To ensure that only feasible rule combinations, i.e., those that correspond to derivations, are valid test cases, we use constraints that exclude infeasible rule combinations. The test cases defined by this specification are in a one-to-one relation to the derivations of length k. We developed an algorithm that calculates the CIT specification for a context-free grammar and a derivation length k. The algorithm determines value sets as well as constraints. This approach enables us to apply a constraint capable CIT generation algorithm to generate a test set that fulfills a required rule coverage criterion. The generated test sets avoid redundancy in test cases by reducing the number of common subderivations. Our evaluation results with grammars derived from a compiler case study demonstrate the remarkable potential of our approach for restricting the test sets in a systematic way.

This paper is organized as follows. In the next section we present definitions and background on context-free grammars and combinatorial interaction testing. In Section III we introduce our approach for generating a CIT specification corresponding to a set of derivations of a grammar. In Section IV we present our algorithm that implements the approach. In Section V we report on evaluation results. Section VI summarizes related work. We conclude in Section VII and present directions of our future work.

II. Definitions and Background

In this section the necessary background of our approach is introduced. We start with the fundamentals on context-free grammars. Subsequently, we introduce the necessary definitions for combinatorial interaction testing.

A. Context Free Grammars

A context free grammar is defined as a tuple $G = (N, \Sigma, P, S)$. Its components are defined as follows:

- N is a finite set of *nonterminal symbols*.
- Σ is a finite set of *terminal symbols* that represent the elementary symbols of the language defined by the grammar. Σ is disjoint from N.

Table I
SET OF PRODUCTION RULES OF THE EXAMPLE GRAMMAR

Rule	ID	Rule
1.	def	$\epsilon \rightarrow$ `Assign((ObjectAddr a), r)`
2.	add	`r` $\rightarrow$ `Plus(r, r)`
3.	addimm	`r` $\rightarrow$ `Plus(r, imm)`
4.	use	`r` $\rightarrow$ `Content(ObjectAddr a)`
5.	r2c	`r` $\rightarrow$ `Const c`
6.	r2i	`r` $\rightarrow$ `imm`
7.	i2c	`imm` $\rightarrow$ `Const c`

- P is a finite set of *production rules*. Each element of P has the form $V \rightarrow w$ with $V \in N$ and $w \in (\Sigma \cup N)^{*}$[1]. V is the left-hand side, w the right-hand side of the rule.
- S is a distinguished nonterminal symbol, the *start symbol*.

The language of a grammar $L(G)$ contains all those strings that can be generated by beginning with the start symbol and repeatedly applying the production rules. The possible *rule applications* define a binary relation $\Rightarrow_G$ on $(\Sigma \cup N)^* \times (\Sigma \cup N)^*$ with $x \Rightarrow_G y$ iff $\exists u, v, p, q \in (\Sigma \cup N)^* : x = upv \wedge y = uqv \wedge p \rightarrow q \in P$. The relation $\Rightarrow_G^*$ on $(\Sigma \cup N)^* \times (\Sigma \cup N)^*$ is defined as the reflexive, transitive closure of $\Rightarrow_G$. A *derivation* for V and s with $V \in N$ and $s \in (\Sigma \cup N)^*$ is a sequence of rule applications that transforms the nonterminal symbol V into the string s, i.e., $(V, s) \in \Rightarrow_G^*$. Its *length* is the number of rules application steps. It is determined by giving for each step the rule applied in that step and the occurrence of its right hand side nonterminal to which it is applied. In a *leftmost derivation*, it is always the leftmost nonterminal that is replaced. A leftmost derivation is determined by the sequence of applied rules only. In the following, the term derivation always means leftmost derivation. A derivation for V and s is *terminal* if $s \in \Sigma^*$, i.e, s contains no nonterminal symbol. The language of the grammar is the set of terminal derivations for the start symbol S. We call rules with the start symbol as left-hand side *initial rules* because they represent the initial of a derivation. A rule with no nonterminals in its right-hand side is called *terminal rule*.

A simple grammar for assignments with variables, constants and addition expressions as right-hand side expressions is defined with the set of nonterminals $N = \{\epsilon$, r, imm$\}$, the set of terminals $\Sigma = \{$`Const, ObjectAddr, Content, Plus, Assign`$\}$, the production rules shown in Table I and the start symbol $S = \epsilon$. The grammar has one initial rule (`def`) and three terminal rules (`use, r2c, i2c`). The derivation <def, add, add, r2c, r2c, r2c> generates the expression shown in Figure 1.

.

[1]$*$ is the Kleene star operator.

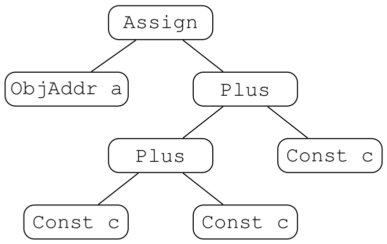


Figure 1. Statement corresponding to the example derivation <def, add, add, r2c, r2c, r2c>.

P1	P2	P3	P4
a_1	b_1	c_1	d_1
	b_2	c_2	d_2
			d_3

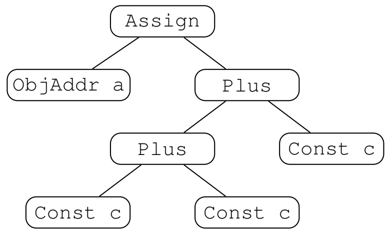
Figure 2. Example CIT Specification with 4 Parameters

In *grammar-based testing*, derivations of a context-free grammar are used as test cases. A tool for grammar-based test data generation performs the reverse task of the compiler front end, i.e., it generates elements of the language defined by the grammar. In general, the generation process is controlled in order to handle the exponential growth of the set of derivations.

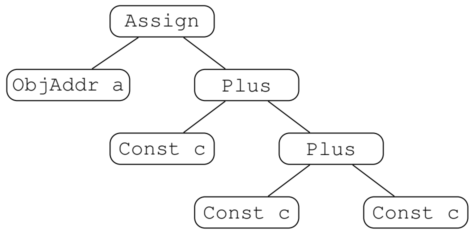
B. Combinatorial Testing

Combinatorial Interaction Testing is a testing approach for detecting failures caused by certain combinations of components or input values. The tester identifies the relevant test aspects and for each test aspect a corresponding class of values. The test aspects are called *parameters*, their elements are called *values*. We define it formally as a set of parameters $P = \{P_1, \ldots, P_n\}$ with each parameter P_i having l_i possible values $P_i = \{p_{i1}, \ldots, p_{il_i}\}$. We assume the parameters to be disjoint sets. A *test case* is a set of n values, one for each parameter. The set of all possible tests is $TS_A = P_1 \times \cdots \times P_n$ and has $\prod_{i=1}^{n} |P_i|$ elements. The notation $|A|$ represents the cardinality of A. The coverage criterion Z of a CIT problem is defined by its strength t. A test set $TS_Z \subseteq TS_A$ fulfills the criterion if every t-wise combination of values from different parameters is covered with at least one test case in TS_Z.

In practice it is rarely the case that the parameters of a system are independent. Constraints describe value combinations that should be excluded for some reason and therefore must not appear in any of the test cases in the calculated result set. The constraints corresponding to a CIT problem can be described with propositional logic formulas. Let $V = \cup_{i=1}^{n} P_i$ be the set of all possible input values. Let A be a corresponding set of atomic propositions with one atomic proposition for each element in V. A constraint c can be described with a Boolean formula over A. A set



(a) Derivation 1



(b) Derivation 3

Figure 3. Statements corresponding to Derivations 1 and 3 from Figure 4.

Table II
TEST SET FOR STRENGTH $t = 2$ FOR EXAMPLE SPECIFICATION

	P1	P2	P3	P4
1	a1	b1	c1	d1
2	a1	b2	c2	d3
3	a1	b1	c2	d2
4	a1	b2	c1	d2
5	a1	b1	c1	d3
6	a1	b2	c2	d1

of forbidden value combinations can invalidate other value combinations. The former are called *explicit* and the latter are called *implicit* constraints. A test case is *valid* if it respects all explicit and implicit constrains, i.e., it does not cover any forbidden value combination.

We illustrate the definitions for CIT with the example specification shown in Figure 2. The corresponding set of all test cases has $12 = 1{*}2{*}2{*}3$ elements. Pairwise coverage, i.e., each possible pair of values from different parameters is covered with at least one test case, can be achieved with the six test cases shown in Table II. The following formula $b_1 \longrightarrow (c_2 \wedge (d_2 \vee d_3))$ ensures that only test cases that combine b_1 with c_2 and d_2 or d_3 are valid. This formula invalidates the test cases 1 and 5. Constraint-capable CIT algorithms ensure that only valid test cases are generated.

III. CIT AND GRAMMAR-BASED TESTING

Our approach exploits the fact that the set of derivations with a fixed length k can be described as a CIT test specification. In this section we explain this observation in detail and introduce the concepts necessary for understanding our approach.

A. CIT Specifications for Derivation Sets

The infinite set of terminal derivations for a grammar can be partitioned into finite sets by the length of the derivations. If we look closely at the set of derivations with length k, we see that many derivations share common subderivations. These common subderivations yield equal substrings in the result of the derivation. Fig. 4 shows some of the terminal derivations with length 6 for our example grammar given in Table I. The columns correspond to the steps, the rows to the derivations. The first and third derivation lead to

	Derivation Step					
	1	**2**	**3**	**4**	**5**	**6**
1	defpsr	add	**add**	**r2c**	**r2c**	r2c
2	defpsr	add	usepsr	add	r2c	r2c
3	defpsr	add	r2c	**add**	**r2c**	**r2c**
...						
6	defpsr	add	addimm	r2c	i2c	usepsr
7	defpsr	add	addimm	usepsr	i2c	usepsr
...						
30	defpsr	add	usepsr	add	usepsr	usepsr

Figure 4. Principle of Parameter Generation for CIT Specification Generation

the two statements shown in Fig. 3, which are symmetric with respect to their right subexpressions. This common subexpression is derived from the highlighted subsequence of rules that appears in both derivations. Assuming that a failure is triggered by the subexpression regardless of its embedding statement, it is reasonable to consider only one of these statements as test case. To exclude such *redundant* statements as test cases, we propose to use combinatorial interaction testing. We generate a CIT specification for a set of terminal derivations with a fixed length k. This enables us to employ CIT test set generation algorithms that aim at generating test sets with small size. The generated test set ensures rule combination coverage with only a small proportion of all test cases generated with k derivation steps.

The CIT specification generated for the set of terminal derivations with length k has k parameters. The values of parameter i are those rules, that are applied in step i of any derivation with length k. This is best illustrated with the derivations shown in Fig. 4. Each column yields a parameter of the CIT specification. The value set associated with a parameter is derived from the content of the column. To avoid invalid rule combinations in a sequence, i.e., those that do not occur together in any terminal derivation, we generate constraints, that exclude these combinations.

The generation of the specification is based on a compact representation for the set of terminal derivations. This representation allows for the efficient generation of the CIT specification. In the following, we explain our compact representation and its generation.

B. Compact Representation of Derivations

For realistic grammars it is very inefficient to enumerate all derivations with a systematic generation process in order to determine the CIT specification because the number of derivations grows exponentially. We avoid the explicit enumeration with a compact representation of all derivations up to the considered length. The construction of this representation is based on the following concepts. A derivation $d =< r_1, r_2, \cdots, r_n >$ with length n can be decomposed into a **prefix** and a **suffix**. The prefix is the first rule r_1 of d, the suffix the remainder of the sequence $< r_2, \cdots, r_n >$. The suffix can be decomposed into subsuffixes by using the nonterminals $nt_1, \cdots, nt_k$ that appear in the right-hand side of the first rule r_1. The following presentation illustrates this decomposition.

$$< \underbrace{r_2, \cdots, r_{i_{nt_1}}}_{nt_1}, \underbrace{r_{i_{nt_1}+1}, \cdots, r_{i_{nt_2}}}_{nt_2}, \cdots, \underbrace{r_{i_{nt_{n-1}}+1}, \cdots, r_n}_{nt_m} >$$

A subsuffix i corresponds to the derivation that transforms the i'th nonterminal into a substring of the overall derivation result. The described decomposition can be applied to each subsuffix in turn.

We illustrate this concept with our example derivation introduced at the end of Section II-A. The prefix of the derivation is the rule `defpsr`, the suffix is the rule sequence <add, add, r2c, r2c, r2c>. This suffix yields the right subexpression of the generated statement. The suffix is not further split by the prefix rule because the rule `defpsr` has only one nonterminal. Subsequently, we decompose this suffix into a further prefix `add` and a further suffix <add, r2c, r2c, r2c>. This suffix is split into the two subsuffixes <add, r2c, r2c> and <r2c>. The first subsuffix yields the left subexpression of the upper `Plus` expression, i.e., the lower `Plus` expression, the second subsuffix the right subexpression of the upper `Plus`.

Inverting the recursive decomposition process leads to a construction procedure, that calculates terminal derivations with increasing length from shorter derivations. During the construction, we can avoid the recomputation of shorter derivations by storing them in a compact form. This compact form is a table that stores for each calculated derivation its length l, its initial nonterminal, its prefix rule and a set of *extended nonterminal patterns* that represent different derivations initializing with rule r and having length $l - 1$. The *extended nonterminal pattern* for a rule $r = nt \rightarrow t^* \, nt_1 \, t^* \cdots t^* \, nt_k \, t^*$ and a derivation $d =< r_1, r_2, \cdots, r_n >$ has the form $((nt_1, l_{nt_1}), \cdots, (nt_k, l_{nt_k}))$. Basically, it is the right-hand side of the rule with all terminal symbols deleted. Additionally, it contains for each nonterminal the length of the corresponding subsuffix. This length is the length of the subderivation that begins at this nonterminal. A rule can be extended by different derivations of the same length because for each of the nonterminals in its right-hand side, there might be derivations of different length that can be combined. The different combinations of the derivations for the nonterminals are represented by different extended nonterminal patterns.

L	NT	Rule	Extended Nonterminal Pattern
1	r	usepsr	$\emptyset$
		r2c	$\emptyset$
	imm	i2c	$\emptyset$
2	r	r2i	< (1, imm) >
	ϵ	defpsr	< (1, reg) >
3	r	addimm	< (1, reg), (1, imm) >
		add	< (1, reg), (1, reg) >
	ϵ	defpsr	< (2,reg) >
4	r	addimm	< (2, reg), (1, imm) >
		add	< (1, reg), (2, reg) >
			< (2, reg), (1, reg) >
	ϵ	defpsr	< (3,reg) >

Figure 5. Compact Representation of Terminal Derivations

In Fig. 5, we illustrate our data structure for our example grammar introduced in Table I. The first three rows of the compact representation table, correspond to the three terminal derivations of length 1. We can derive a string of only terminal symbols from the nonterminal r by using the rules `usepsr` and `c2r`. From the nonterminal imm we derive only one such string by applying the terminal rule `c2i`. Since the start symbol is not stored for length 1, we know that it is not possible to derive an element of the language with only one step. The following two lines represent three terminal derivations of length 2. The nonterminal r yields only one terminal derivation, but the extended nonterminal pattern for the rule `defpsr` can be combined with both alternatives for the nonterminal r with length 1. These two combinations represent the two possible elements of the language that can be generated with two rule application steps. We see in our table, that for length 4, nonterminal r and rule `add` there are two different extended nonterminal patterns. These different extended nonterminal patterns reflect the symmetry that we exemplified with the two statements in Fig. 3. We can apply the shorter subderivation generating the constant expression either on the left-hand side nonterminal or on the right-hand side nonterminal.

The first three rows of the table, correspond to the three terminal derivations of length 1. Remember that a terminal derivation yields a string of only terminal symbols and must not initialize with the start symbol of the grammar. We can derive a string of only terminal symbols from the nonterminal r by using the rules `usepsr` and `r2c`. From the nonterminal imm we derive only one such string by applying the terminal rule `i2c`. Since the start symbol is not stored for length 1, we know that it is not possible to derive an element of the language with only one step. The following two lines represent three terminal derivations of length 2. The nonterminal r yields only one terminal derivation because the nonterminal imm in its extended nonterminal pattern can be combined with only the third row of the table. The extended nonterminal pattern for the rule `defpsr` can be combined with both alternatives for the nonterminal r with length 1. These two combinations represent the two possible elements of the language that can be generated with two rule application steps. We see in our table, that for length 4, nonterminal r and rule `add` there are two different extended nonterminal patterns. These different extended nonterminal patterns reflect the symmetry that we exemplified with the two statements in Fig. 3. We can apply the shorter subderivation, which generates the constant expression, either on the left-hand side nonterminal or on the right-hand side nonterminal.

IV. Our Algorithm

A. Calculation of Terminal Derivations

Our algorithm that calculates the compact representation for all terminal derivations with a fixed length is given in Listing 1. The top level procedure `calculateDTable` solves two tasks. First, it initializes the first row of the table with all terminal rules, i.e., all rules that have no nonterminal in their right hand side. Second, it calls the procedure `calculateTDerivations` to iteratively determine the derivations with length 2 up to the length specified by the parameter.

The procedure `calculateTDerivations` is shown in Listing 1 below the top-level function. This procedure extends the table with all nonterminal/rule/set of extended nonterminal patterns combinations that correspond to a terminal derivation with the required length. The procedure considers successively all rules of the grammar. If a rule is terminal, it can not be used as a prefix of a derivation and thus can be skipped. In the other case, we calculate the set of extended nonterminal patterns with the procedure `calculateExtNTPatterns` and store them in the table.

The procedure `calculateExtNTPatterns` calculates for a rule and a length the set of extended nonterminal patterns representing a derivation with the given length. The idea is to split up the given length for all nonterminals. There can be several such splits. Each split is a possible pattern candidate. A candidate $(l_1, \cdots, l_k)$ for a rule r with $nt_1, \cdots, nt_k$ nonterminals in its right hand side corresponds to a terminal derivation if there is for the i'th nonterminal a terminal derivation with length l_i. First, our algorithm determines the nonterminal pattern of the considered rule and its number of elements `k`. Subsequently, we calculate all possible splits of the parameter $length$ into k integers $i_1, \cdots, i_k$. Each split must be validated. To validate a split,

```
calculateDTable(length, grammar){
  foreach(rule in grammar){
    if(isTerminalRule(rule)){
      updateTable(table, 1, getLhs(r), getID(r), null)
    }
  }
  for(i=2 to length){
    calculateTDerivations(i, table, grammar)
  }
}
calculateTDerivations(length, table, grammar){
  foreach(rule in grammar){
    if(!isTerminal(rule)){
      decompositions = calculateExtNTPatterns(table,
                                              rule, length−1)
      //consider only if at least one derivation with length exist
      if(notEmpty(decompositions)){
        updateTable(table, length,
                    getLhs(r), getID(r), decompositions)
      }
    }
  }
}
calculateExtNTPatterns(table, rule, length){
  NTPattern = getNonterminalPattern(rule)
  k = numberOfElements(NTPattern)
  allSplits = calculateSplits(length, k)
  init(result)
  foreach(split in allSplits){
    init(extendedNTPattern)
    isValid = true
    //verify all sublengths
    for(i=0 to k−1){
      nt = NTPattern[i]
      lengthForNT = split[i]
        if(!inTable(table, lengthForNT, nt){
          isValid = false; break;
        }else{
          append(extendedNTPattern, nt, i)
        }
    }
    if(isValid){
      append(result, extendedNTPattern)
    }
  }
  return result
}
```

Listing 1. Calculation of Compact Representation for Terminal Derivations

it must be checked if there is a derivation for the nonterminal nt_i with length i. In case that there is no such terminal derivation the split is invalid. In case that for all nonterminals there is a terminal derivation with the length given by the split, it is valid. The check of all splits is implemented by the `foreach-loop`, the validation of an individual split by the inner `for-loop`. The auxiliary procedure `inTable` validates that a nonterminal-length combination exists in the table. In this case, there exists at least one terminal derivation with the desired length that begins at the given nonterminal.

To illustrate our algorithm, we explain the calculation

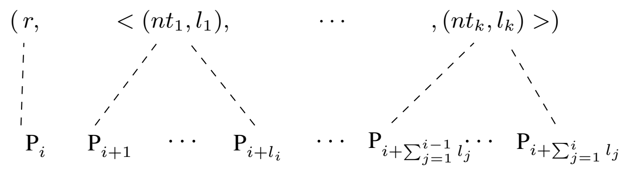


Figure 6. Calculation of Value Sets for Extended Nonterminal Pattern

of the fourth row of the table shown in Fig. 5. First, the algorithm has determined the rows one to three with the for-loop in the top-level procedure. To calculate the fourth row, we consider all rules of the grammar that are not terminal. This is done with the foreach-loop in procedure `calculateTDerivations`. We consider the rules `defpsr, add, addimm` and `r2i`. Most interesting is the calculation for the rule `addimm`. Its right-hand side consists of two nonterminals (r, imm), i.e., we must split the length 3 for the suffix into two values. Remember that the derivation has length 4 and its first element is the rule `addimm` itself. There are two possible splits, i.e., (1,2) and (2,1). Only the first split is valid. There is no row with length 2 and nonterminal imm in our table, i.e., there is no terminal derivation of length 2 that initializes with this nonterminal. Consequently, the second split is invalid for this rule. For the rule `add` both splits are valid.

B. CIT Specification Calculation

Our algorithm `calculateCITSpec` that generates the CIT specification corresponding to all terminal derivations with length l is shown in Listing 2. It consists of three steps. First, it calls the previously presented procedure `calculateDTable` that builds the compact representation. Subsequently, the value sets of the specification are calculated by the procedure `calculateParams`. Finally, the constraints are generated by the procedure `calculateConstraints`.

Basically, both procedures `calculateParams` and `calculateConstraints` recursively traverse the table. The traversal begins with rows that represent long derivations and descends to rows that represent short derivations. The traversal process is controlled by the length values and the nonterminals in the extended nonterminal patterns. During the traversal process, the rule identifiers associated with the extended nonterminal patterns are inserted into the parameter corresponding to the depth of the descent by the procedure `calculateParams`. The procedure `calculateConstraints` combines them into a logical formula. Fig. 6 shows how the nonterminal/length pairs in an extended nonterminal pattern are related to the parameters. Given that the prefix rule is applied in step i it must belong to the parameter i. The following parameters must be extended as indicated by the nonterminal/length pairs of the extended nonterminal pattern of the rule. We start a traversal in a row representing all terminal derivations of

```
**calculateCITSpec**(length, grammar){
  cit = initCITSpec(length)
  table = calculateDTable(length, grammar)
  **calculateParams**(cit, length, table, getStartsymbol(grammar))
  **calculateConstraints**(cit, length, table, getStartsymbol(grammar))
}
// getDerivations returns
// [(RID_1, extendedNTPattern_1, ..., extendedNTPattern_{m_1}),
// ..., (RID_n, extendedNTPattern_1, ..., extendedNTPattern_{m_n})]
**calculateParams**(cit, length, table, startsymbol){
  allDerivations = getDerivations(table, length, startsymbol)
  **foreach**(derivationsForRule in allDerivations){
    **calculateCITValues**(table, cit, 1, derivationsForRule)
  }
}
// derivationsForRule = (RID, extendedNTPattern_1,
// ..., extendedNTPattern_m)
// extendedNTPattern = ( (l_1, nt_1), ..., (l_k, nt_k))
**calculateCITValues**(derivationsForRule, cit, idx, table){
  updateParameter(cit, idx, getRuleID(derivationsForRule))
  extendedNTPatterns = getExtNTPatterns(derivationsForRule)
  **foreach**(extendedNTPattern in extendedNTPatterns){
    insertIDX = idx + 1;
    **foreach**(lengthNTPair in extendedNTPattern){
      length = getLength(lengthNTPair)
      nt = getNonterminal(lengthNTPair)
      **if**(1 == length){
        RIDS = getRuleIDs(table, length, nt)
        updateParameter(cit, insertIDX, RIDS)
      }
      **else**{
        allDerivations = getDerivations(table, length, nt)
        **foreach**(derivationsForRule in allDerivations){
          **calculateCITValues**(table, cit, insertIDX,
                                 derivationsForRule)
        }
      }
      insertIDX = idx+length
    }
  }
}
```

Listing 2. CIT Specification Calculation

length l that begin with the nonterminal nt. We successively consider all rules that are stored in our table for length l and nonterminal nt. For each rule, we must take into account all its associated extended nonterminal patterns. For each extended nonterminal pattern to be considered, the rows to be visited next are indicated by its nonterminal/length pairs. For each nonterminal/length pair with length greater than 1 we descend further with a recursive call. In case that the length is 1 we have reached the base case and return.

First, we present the algorithm that determines the value sets for the parameters, subsequently we present the algorithm that calculates the constraints.

Calculation of Parameters and Values: The procedure `calculateCITValues` given in Listing 2 below the top-level procedure extends the specification to be constructed with a given rule identifier and the rules that can be used in derivations indicated by the extended nonterminal patterns associated with the rule identifier. If a rule identifier is added to the parameter representing the i'th derivation step, the extended nonterminal pattern determines the rule identifiers that must be added to the subsequent parameters. In Fig. 6 this functionality is illustrated. It is shown how the rule identifier and the nonterminal/length pairs of an associated extended nonterminal pattern are assigned to the parameters. The rule identifier is added to the parameter indicated by the index that is passed to the function as parameter. The nonterminal/length pairs yield the values for the succeeding parameters. After processing a nonterminal/length pair, with the traversal explained above, the index that indicates the parameter to be extended next is updated by using the length value in the pair. The rule identifier and all its nonterminal patterns are given by the parameter `derivationsForRule`. The second parameter represents the CIT specification to be build. In the first step, the procedure adds the rule identifier to the parameter indicated by the index parameter `idx`. Subsequently, the extended nonterminal patterns are considered successively to extend the following parameters. This is implemented by the outer `foreach-loop`. Before we start the iteration for an extended nonterminal-pattern, we reset the index for the nonterminal-pattern under consideration. Subsequently, we iterate over the nonterminal/length pairs with the following `foreach-loop`. In case that the nonterminal/length pair has length 1 no recursive call is necessary. We determine the corresponding rule identifiers for the nonterminal from our table and add them to the parameter. These rule identifiers belong to terminal rules that have the nonterminal pattern as left-hand side. In case that the length l is greater than 1, we request all rows with the length l and the considered nonterminal by using the auxiliary function `getDerivations`. These rows represent the terminal derivations of length l that begin with the considered nonterminal. For each delivered row, the function is called recursively. This is implemented by the `foreach-loop` in the else-branch. After consideration of a nonterminal/length pair, the index indicating the parameter to be extended next is increased by the length l. The procedure `calculateParams` that is given below the procedure `calculateCITValues` initializes the traversal process for all terminal derivations that begin with the start symbol of the grammar and have the required length. Our goal is to generate a CIT specification that corresponds to a set of terminal derivations that generate elements of the language. For this reason, we restrict the generation process to derivations beginning with the start symbol.

Calculation of Constraints: The previously presented procedure generates a CIT specification that may allow for test cases that do not correspond to terminal derivations. To exclude invalid combinations of rules, we generate an individual constraint for each initial rule that yields at least one element of the language with the considered number of derivation steps. The construction of constraints is based on observations that we explain in the following. First, we consider a row of our table that contains for the required length an initial rule, i.e., a rule that has the start symbol as its left-hand side pattern.

$$\begin{array}{ll} (start, l, r) & <(nt_1, l_{11}), \cdots, (nt_k, l_{1k_1})>, \\ & \cdots \\ & <(nt_1, l_{1m}), \cdots, (nt_k, l_{1k_m})> \end{array}$$

We know that any derivation beginning with our rule continues with one of the derivations determined by the associated extended nonterminal patterns. The formula representing the constraint for our initial rule r is an implication constructed with subformulas f_i that represent the individual extended nonterminal patterns. These subformulas are combined into a disjunction as it is shown below. In case that we select an initial rule different from the rule r the constraint is trivially fulfilled, without taking into account the subformulas for the extended nonterminal patterns. Otherwise, at least one subformula must evaluate to true.

$$r \longrightarrow (f_1 \vee \cdots \vee f_m)$$

Now we consider an extended nonterminal pattern. Its form is shown below:

$$<(nt_1, l_1), \cdots, (nt_k, l_k)>$$

Given that $f_1, \cdots, f_k$ are the subformulas for the nonterminal/length pairs, the formula for the nonterminal pattern is a conjunction of all subformulas f_i as shown below. We require that all subformulas that result from the subderivations for the single nonterminals are fulfilled.

$$f_1 \wedge \cdots \wedge f_k$$

The formula for a rule r' that is applied in an intermediate step i is similar to a formula for an initial rule. It is also constructed from subformulas $f_1, \cdots, f_m$ for all its extended nonterminal patterns. It differs from the formula for an initial rule by the implication. The implication is substituted with the $\wedge$ operator as it is shown below.

$$r' \wedge (f_1 \vee \cdots \vee f_m)$$

We cannot construct the formula for such a rule with the implication operator. If we select another rule that can also be used in step i the constraint would be trivially fulfilled, even if this is not really the case. This is best illustrated with a small example. Given a very small grammar that yields the following derivations of length 3 with the rules $r_{s_1}, r_{s_2}, r_{21}, r_{22}, r_{23}, r_{33}, r_{34}$.

1	r_{s_1}	r_{21}	r_{31}
2	r_{s_1}	r_{21}	r_{32}
3	r_{s_1}	r_{22}	r_{33}
4	r_{s_2}	r_{23}	r_{34}

We know that in any derivation with length 3, there is only one element of the following subsets selected

$\{\{r_{s_1}, r_{s_2}\}, \{r_{21}, r_{22}, r_{23}\}, \{r_{31}, r_{32}, r_{33}, r_{34}\}\}$. Our constraints are thus implicitly combined with a formula of the form

$$((r_{s_1} \wedge \neg r_{s_2}) \vee (\neg r_{s_1} \wedge r_{s_2})) \bigvee$$
$$((r_{21} \wedge \neg r_{22} \wedge \neg r_{23}) \vee (\neg r_{21} \wedge r_{22} \wedge \neg r_{23}) \vee (\neg r_{21} \wedge \neg r_{22} \wedge r_{23})) \bigvee$$
$$\ldots$$

Given that our constraints are defined as shown below, we cannot ensure that r_{22} is combined with r_{32} only.

$$\begin{aligned} c_{s_1}: \quad & r_{s_1} \longrightarrow ((r_{21} \longrightarrow (r_{31} \vee r_{32})) \vee (r_{22} \longrightarrow r_{33})) \\ c_{s_2}: \quad & r_{s_2} \longrightarrow (r_{23} \longrightarrow r_{34}) \end{aligned}$$

A conjunctive formula of the above shown *implicit formula* for the possible combinations of rules and the constraints evaluates to true for the assignment $r_{s_1} = true$, $r_{22} = true$ and $r_{31} = true$ and for all other values $false$. The corresponding test case $(r_{s_1}, r_{22}, r_{31})$ is not valid. By substituting the *inner* implication with a conjunction as explained before, we achieve the intended behavior.

Our algorithm `calculateCITConstraints` that generates the constraints is shown in Listing 3. It generates the described formulas by also using our compact representation. The data structures and auxiliary functions used in the presentation have been introduced in Listing 2. In order to calculate the constraints, first we determine all rows of our table that represent terminal derivations with an initial rule as first rule. For each such rule, we construct the disjunction of formulas for the alternative extended nonterminal patterns associated with it. These formulas are calculated by the two mutually recursive procedures `formulaForNTPattern` and `formulaForLengthNTPair` shown in the same listing.

The procedure `formulaForNTPattern` constructs the conjunctive formula corresponding to a nonterminal pattern. It calls for each nonterminal/length pair the procedure `formulaForLengthNTPair`. In case that the parameter `lengthNTPair` passed to the procedure `formulaForLengthNTPair` contains a length equal to 1 no further recursive descent is required. We extract from our table all rule identifiers of terminal rules that have the nonterminal symbol contained in the parameter as left-hand side. We combine them into a disjunctive formula. In case that the length is greater than 1 we determine the set of derivations for the nonterminal/length pair. For each extended nonterminal pattern in such combination we call the procedure `formulaForNTPattern`. The resulting formulas are combined into a disjunction. Finally, we generate the conjunction of the rule identifier appearing in the combination and the generated disjunction.

Once we have generated our CIT specification, we can employ a constraint capable CIT test set generation algorithm to generated a set of terminal derivations with length k.

```
calculateCITConstraints(cit, length, table, startsymbol){
  allDerivations = getDerivations(table, length, startsymbol)
  foreach(derivationsForRule in allDerivations){
    rid = getRuleID(derivationsForRule)
    extendedNTPatterns = getExtNTPatterns(derivationsForRule)
    foreach(extendedNTPattern in extendedNTPatterns){
      f = f $\vee$ formulaForNTPattern(extendedNTPattern)
    }
    formula = rid $\longrightarrow$ f
    addConstraint(cit, formula)
  }
}
formulaForNTPattern(extendedNTPattern, table){
  foreach(lengthNTPair in extendedNTPattern){
    f = formulaForLengthNTPair(lengthNTPair, table)
    result = result $\wedge$ f
  }
  return result
}
formulaForLengthNTPair(lengthNTPair, table){
  length = getLength(lengthNTPair)
  nt = getNonterminal(lengthNTPair)
  if(1 == length){
    RIDS = getRuleIDs(table, length, nt)
    foreach(rid in RIDS){
      result = result $\vee$ rid
    }
  }
  else{
    allDerivations = getDerivations(table, length, nt)
    foreach(derivationsForRule in allDerivations){
      rid = getRuleID(derivationsForRule)
      extendedNTPatterns = getExtNTPatterns(derivationsForRule)
      foreach(extendedNTPattern in extendedNTPatterns){
        f = f $\vee$ formulaForNTPattern(extendedNTPattern, table)
      }
      result = rid $\wedge$ f
    }
  }
  return result
}
```

Listing 3. Constraint Calculation

Depending on the strength t of the interaction, it is ensured, that each t-wise rule combination is covered at least once. Because CIT test set generation aims at calculating small sets, the number of redundant subderivations is reduced. In the following, we present our evaluation results that demonstrate the remarkable potential of our approach.

V. Evaluation

In this section, we provide evaluation results for grammars from our compiler case study. The grammar that we used for our evaluation is taken from a compiler back end specification for the Intel Itanium [3]. From this specification the compiler module is generated that translates the internal program representation of the compiler into the assembly language. A good introduction into this technique is given in [4]. The Itanium compiler is build with the CoSy

Table III
EVALUATION RESULTS FOR CIT SPECIFICATION GENERATION

		P_1	P_2	P_3	P_4	P_5	P_6
G1	Spec3	5	17	9			
	Spec4	10	38	26	13		
	Spec5	10	43	38	26	13	
	Spec6	12	55	47	38	26	13
G2	Spec3	5	21	13			
	Spec4	10	44	30	20		
	Spec5	10	53	47	32	20	
	Spec6	12	88	67	54	32	20

Table IV
EVALUATION RESULTS FOR TEST SET CALCULATION FOR PAIRWISE RULE COVERAGE BY CIT.

		G1	G2	G3	G4
L		1S,4E	2S, 4E	1S, 5E	2S, 5E
4	Spec	$1\ 2^3$	2^4	$1\ 3\ 2^2$	$2^3\ 3$
	Derivations	8	16	12	24
	TCases(PICT)	4	6	5	6
	TCases(ACTS)	4	6	6	6
6	Spec	$1\ 2^3\ 4^2$	$2^4\ 4^2$	$1\ 2^2\ 3\ 5^2$	$2^3\ 3\ 5^2$
	Derivations	64	128	144	288
	TCases(PICT)	13	13	20	19
	TCases(ACTS)	16	25	13	22
8	Spec	$1\ 2^3\ 4^4$	$2^4\ 4^4$	$1\ 2^2\ 3\ 5^4$	$2^3\ 3\ 5^4$
	Derivations	640	1280	2160	4320
	TCases(ACTS)	22	32	17	28
10	Spec	$1\ 2^3\ 4^6$	$2^4\ 4^6$	$1\ 2^2\ 3\ 5^6$	$2^3\ 3\ 5^6$
	Derivations	7168	14336	36288	-
	TCases(ACTS)	32	38	24	30

compiler development system [5]. The grammar defined in the specification generates the statements and expressions of the intermediate representation. To test the generated compiler module systematically, we derive single statements. A derived statement is generated within the compiler and embedded into a program frame. The resulting program is translated with the compiler module under test into an executable program and translated back into a source language program in order to apply differential testing as test evaluation method [6]. Differential testing requires a second implementation of the system under test that executes with the same inputs. In our case a second compiler for the source language C is required. We translate the source language program with the second compiler. Subsequently, we execute both test programs, i.e., the binary produced by the compiler back end under test and the binary produced by the test oracle compiler. Finally, we compare their outcomes. In case that the results differ, a failure has been detected, but the failure may also result from the test oracle system.

Our first experiments demonstrate that our test specification generation algorithm is applicable for realistic grammars. Subsequently, we prevent evaluation results for limited subsets of the compiler grammars. These results show that our approach provides a suitable way to restrict the test sets that can be generated for a grammar.

We used in our first experiments two grammars G1 and G2. These grammars are restricted versions of the original grammar. We do not need to consider the complete specification because not all rules are relevant for testing. The first grammar G1 has 86 rules with 12 initial rules, the second grammar G2 has 122 rules with the same number of initial rules. The generation time never exceeds 5 seconds for all specifications. In Table III the evaluation results with respect to the generated specifications are shown. The columns P_1 to P_6 show the number of values for the corresponding parameter. The rows show the generated specifications for derivation lengths between three and six. The number of initial rules is the upper bound for the number of values in the first parameter. Most parameters have more than 10 values. This is in contrast to many benchmark instances for CIT generation algorithms. Frequently used benchmarks are presented in [7]. These benchmarks consider only specifications with at most six values for a parameter. Specifications with many-valued parameters have been used in [8] and [9], but both approaches are not capable of handling constraints.

We conducted a second series of experiments with very limited grammars. There are two reasons for this: the number of available tools for CIT test set generation with constraints is limited and very often the constraints must be given as Boolean formulas in disjunctive normal form. This limitation applies for example to the constraint capable tool CASA [7]. The PICT tool [10] and the ACTS tool [11] support constraints in a less restricted form, but in our experiments they failed to process the instances shown in Table III. We reduced our grammar to the subsets shown in Table IV. The smallest grammar G1 has only five rules, with one initial rule, the biggest grammar has seven rules, with two initial rules. Even though the grammars are quite small, the results are striking. The table presents for the derivation length 4, 6, 8 and 10 the size of the generated specification (Spec). We present the size of the specification in the common notation for CIT specifications, i.e., n^k means that the specification has k parameters with n values. Moreover, we show the number of terminal derivations that can be derived with the given number of derivation steps (Derivations) and the number of test cases required to achieve pairwise rule coverage (Test Cases). We show the results calcuated by PICT and ACTS. PICT failed to process the instances for derivation length bigger than 6.

Our results show that only a small proportion of all test cases that can be derived with a fixed length is required to ensure that all possible pairs of grammar rules are covered. The considered pairs are those that appear in any derivation of the considered length. It would be a huge advantage if the CIT test set generation algorithms would be improved, such that they can handle the CIT specifications that our approach generates. This would allow for a very efficient test set selection technique for grammar-based testing.

VI. Related Work

One of the first approaches for grammar-based testing has been presented in [12]. It generates a test set that covers each rule of the grammar with at least one test case. *Stochastic* reduction techniques for grammar-based techniques annotate the rules with weights that guide the derivation process. This approach has been applied to test compilers [6] and also to test Java virtual machines [1]. It does not aim at a rule coverage criterion, but offers the possibility to prioritize single rules. The reduction approach presented in [2] uses control mechanisms. Most related to our approach is the control mechanism *dependence control*. The authors sketch the general idea but they do not provide information on how to achieve rule coverage. They point out that the introduced algorithm is capable of only all-way and one-way coverage. All-way means an exhaustive enumeration of derivations.

The approach presented in [13] is similar to our approach, i.e., combinatorial interaction testing is used to systematically derive a test set for a compiler. In contrast to our approach, the required specification must be defined manually and is not based on a grammar. In [14] an approach is presented that analyses usage log in order to create test cases. After the log file has been analysed, the created test set can be prioritized by e.g. event interaction. In contrast to this approach, we generate the test set from scratch by calculating a CIT specification from a given grammar.

A good overview on CIT test generation with constraints is given in [15]. We are not aware of a detailed investigation of the capability of constraint capable CIT algorithms with regard to the complexity of the constraints.

Our discussion shows that grammar-based testing can be applied in a broad range of areas. However, it must be pointed out that test selection techniques are required. Different techniques that address this problem have been presented but we are not aware of any approach that uses combinatorial interaction testing to select test cases systematically.

VII. Conclusion

In this paper, we have presented a new approach for grammar-based test set generation. Our approach uses combinatorial interaction testing to determine a set of tests that ensures t-wise coverage of grammar rules. This approach enables us to reduce the redundancy caused by subderivations that appear repeatedly. We have introduced an algorithm that generates for a grammar and a derivation length a test specification for combinatorial interaction testing. This algorithm is based on a compact representation of the set of all relevant derivations. The practical applicability of our specification generation approach is shown by our evaluation results. We used our algorithm to generate test specifications for testing the compiler back end specification of a real compiler. We plan to improve our algorithm with respect to the constraint generation. In particular, we will investigate how *forbidden combinations* can be determined by using our compact derivation. This would enable us to use a broader range of constraint capable tools. Another possible direction for further research is the integration of our approach with control mechanisms for test set reduction.

References

[1] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," in *DSL*, 1999, pp. 1–13.

[2] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," in *TestCom*, 2006, pp. 19–38.

[3] L. Gesellensetter and S. Glesner, "Interprocedural speculative optimization of memory accesses to global variables," in *Euro-Par*, 2008, pp. 350–359.

[4] A. Nymeyer, J.-P. Katoen, Y. Westra, and H. Alblas, "Code Generation = A* + BURS," in *CC*, 1996, pp. 160–176.

[5] Associated Compiler Experts bv., Amsterdam, The Netherlands, `http://www.ace.nl`.

[6] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Trans. Software Eng.*, vol. 34, no. 5, pp. 633–650, 2008.

[8] A. Calvagna and A. Gargantini, "IPO-s: Incremental generation of combinatorial interaction test data based on symmetries of covering arrays," in *ICSTW*. IEEE, 2009, pp. 10–18.

[9] E. Salecker and S. Glesner, "Pairwise test set calculation using k-partite graphs," in *ICSM*, 2010, pp. 1–5.

[10] J. Czerwonka, "Pairwise testing in real world," in *Proceedings of 24th Pacific Northwest Software Quality Conference*, 2006.

[11] R. D. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, 2008.

[12] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, vol. 12, no. 3, pp. 366 – 375, 09 1972.

[13] R. Mandl, "Orthogonal latin squares: an application of experiment design to compiler testing," *Commun. ACM*, vol. 28, no. 10, pp. 1054–1058, 1985.

[14] R. C. Bryce, S. Sampath, and A. M. Memon, "Developing a single model and test prioritization strategies for event-driven software," *IEEE Trans. Softw. Eng.*, vol. 37, pp. 48–64, 2011.

[15] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2007, pp. 129–139.