Controllable Combinatorial Coverage in Grammar-Based Testing

Ralf Lämmel[1] and Wolfram Schulte[2]

[1] Microsoft Corp., Webdata/XML, Redmond, USA
[2] Microsoft Research, FSE Lab, Redmond, USA

Abstract. Given a grammar (or other sorts of meta-data), one can trivially derive combinatorially exhaustive test-data sets up to a specified depth. Without further efforts, such test-data sets would be huge at the least and explosive most of the time. Fortunately, scenarios of grammar-based testing tend to admit non-explosive approximations of naive combinatorial coverage.

In this paper, we describe the notion of controllable combinatorial coverage and a corresponding algorithm for test-data generation. The approach is based on a suite of control mechanisms to be used for the characterization of test-data sets as well-defined and understandable approximations of full combinatorial coverage.

The approach has been implemented in the C#-based test-data generator *Geno*, which has been successfully used in projects that required differential testing, stress testing and conformance testing of grammar-driven functionality.

1 Introduction

This paper is about *grammar-based testing* of software. We use the term 'grammar' as a placeholder for context-free grammars, algebraic signatures, XML schemas, or other sorts of meta-data. The system under test may be a virtual machine, a language implementation, a serialization framework for objects, or a Web Service protocol with its schema-defined requests and responses. It is generally agreed that manual testing of grammar-driven functionality is quite limited. *Grammar-based test-data generation* allows one to explore the productions of the grammar and grammatical patterns more systematically. The test-oracle problem has to be addressed in one of two ways: either multiple implementations are subjected to differential testing (e.g., [20]), or the intended meaning of each test case is computed by an extra model (e.g., [23]).

Prior art in grammar-based testing uses *stochastic test-data generation* (e.g., [19, 20, 23]). The canonical approach is to annotate a grammar with probabilistic weights on the productions and other hints. A test-data set is then generated using probabilistic production selection and potentially further heuristics. Stochastic approaches have been successfully applied to practical problems. We note that this approach requires that coverage claims are based on stochastic arguments. In our experience, the actual understanding of coverage may be

M.Ü. Uyar, A.Y. Duale, and M.A. Fecko (Eds.): TestCom 2006, LNCS 3964, pp. 19–38, 2006.

challenging due to intricacies of weights and other forms of control that 'feature-interact' with the basic stochastic model.

The work reported in this paper adopts an alternative approach to test-data generation. The point of departure is full *combinatorial coverage* of the grammar at hand, up to a given depth. Without further efforts, such test-data sets would be huge at the least and explosive most of the time. Hence, approximations of combinatorial coverage are needed. To this end, our approach provides *control mechanisms* which can be used in modeling the test problem. For instance, one may explicitly limit the recursive applications for a given sort ('nonterminal')[1], and thereby scale down the 'productivity' of that sort. The control mechanisms are designed in such a way that the approximations of combinatorial coverage are intelligible. In particular, the effect of each use of a mechanism can be perceived as a local restriction on the operation for term construction.

The approach has been implemented in the C#-based test-data generator *Geno*.[2] The input language of *Geno* is a hybrid between EBNF and algebraic signatures, where constructors and sorts can be annotated with control parameters. *Geno* has been successfully used in development projects over the last 2+ years at Microsoft. These projects required differential testing, stress testing and conformance testing of grammar-driven functionality.

The paper is structured as follows. The overall approach is motivated and illustrated in Sec. 2. The basics of combinatorial test-data generation are laid out in Sec. 3 – Sec. 5. The control mechanisms are defined in Sec. 6. A grammar-based testing project is discussed in Sec. 7. Related work is reviewed in Sec. 8. The paper is concluded in Sec. 9.

2 Controllable Combinatorial Coverage in a Nutshell

The following illustrations will use a trivial expression language as the running example, and it is assumed that we want to generate test-data for testing a code generator or an interpreter. We further assume that we have access to a test-oracle; so we only care about test-data generation at this point. Using the grammar notation of *Geno*, the expression language is defined as follows:

```
Exp = BinExp ( Exp , BOp, Exp )  // Binary expressions
    | UnaExp ( UOp , Exp )       // Unary expressions
    | LitExp ( Int ) ;           // Literals as expressions

BOp = "+" ;  // A binary operator
UOp = "-" ;  // A unary operator
Int = "1" ;  // An integer literal
```

[1] We use grammar- vs. signature-biased terminology interchangeably. That is, we may say nonterminal vs. sort, production vs. constructor, and word vs. term.

[2] *Geno* reads as "*Gen*erate *o*bjects" hinting at the architectural property that test data is materialized as objects that can be serialized in different ways by extra functionality.

Depth	G_a	G_b	G_c	G_d
1	0	0	0	1
2	1	3	6	29
3	2	42	156	9.367
4	10	8.148	105.144	883.148.861
5	170	268.509.192	–	–
6	33.490	–	–	–
7	–	–	–	–

Fig. 1. Number of terms with the given depth for different grammars ('–' means outside the long integer range 2.147.483.647); G_a is the initial grammar from the beginning of this section; G_b comprises 3 integer literals (0, 1, 2), 2 unary operators ('+', '−'), and 4 binary operators ('+', '−', '∗', '/'); G_c further adds variables as expression form along with three variable names (x, y, z); G_d further adds typical expression forms of object-oriented languages such as C#

We can execute this grammar with *Geno* to generate all terms over the grammar in the order of increasing depth. The following C# code applies *Geno* programmatically to the above grammar (stored in a file `"Expression.geno"`) complete with a depth limit for the terms (cf. 4). The `foreach` loop iterates over the generated test-data set such that the terms are simply printed.

```
using Microsoft.AsmL.Tools.Geno;

public class ExpressionGenerator {
  public static void Main (string[] args) {
    foreach(Term t in new Geno(Geno.Read("Expression.geno"), 4))
      Console.WriteLine(t);
  }
}
```

Let us review the combinatorial complexity of the grammar. We note that:

- there is no term of sort `Exp` with depth 1 (we start counting depth at 1);
- there is 1 term of sort `Exp` with depth 2: `LitExp`("1");
- ... 2 terms ... with depth 3:
  - `UnaExp`("-",`LitExp`("1")),
  - `BinExp`(`LitExp`("1"),"+",`LitExp`("1"));
- ... 10 terms ... with depth 4;
- hence, there are 13 terms of sort `Exp` up to depth 4;
- the number of terms explodes for depth 6 — 7.

In Fig. 1, the number of terms with increasing depth is shown. We also show the varying numbers for slightly extended grammars. We note that all these numbers are about expression terms *alone*, neglecting the context in which such expressions may occur in a non-trivial language. Now suppose that we consider a

grammar which has nonterminals for programs, declarations and statements — in addition to expressions that are used in statement contexts. With full combinatorial exploration, we cannot expect to reach expression contexts and to explore them for some non-trivial depth.

Combinatorial coverage can be *approximated* in a number of ways. One option is to give up on combinatorial completeness for the argument domains when constructing terms. In particular, one could choose to exhaust the argument domains *independently* of each other. Such an approximation is justified when the grammar-driven functionality under test indeed processes the corresponding arguments independently, or when the test scenario is not concerned with the dependencies between the arguments.

In reference to pairwise testing [18] (or two-way testing), we use the term *one-way testing* for testing argument domains independently of each other. Combinatorially exhaustive testing of arguments domains is then called *all-way testing*. In the running example, we want to require one-way testing for the the constructor of binary expressions. Here we assume that the system under test is a simple code generator that performs *independent* traversal on the operands of `BinExp`.

A *Geno* grammar can be directly annotated with control parameters:

```
Exp = [Oneway] BinExp ( Exp , BOp, Exp )
    | UnaExp ( UOp , Exp )
    | LitExp ( Int ) ;
```

Alternatively, one may also collect control parameters in a separate test specification that refers to an existing grammar. The above example is then encoded as follows:

```
[Oneway] Exp/BinExp ;
```

Let us consider another opportunity for approximation. We may also restrict the normal or recursive *depth* of terms on a specific argument position of a specific constructor. By the latter we mean the number of nested applications of recursive constructors. Such an approximation is justified when the grammar-driven functionality under test performs only straightforward induction on the argument position in question, or when the specific test scenario is not concerned with that position. In the running example, we want to limit the recursive depth of expressions used in the construction of unary expressions:

```
[MaxRecDepth = 1] Exp/UnaExp/2 ;
```

Here "2" refers to the 2nd parameter position of `UnaExp`. The helpful effect of the `Oneway` and `MaxRecDepth` approximations is calculated in Fig. 2. We showcase yet another form of control, for which purpose we need to slightly extend the grammar for expressions. That is, we add a nonterminal, `Args`, for sequences of arguments, just as in a method call.

```
Args = (Exp*) ;
```

Grammar	Depth = 1	Depth = 2	Depth = 3	Depth = 4	Depth = 5	Depth = 6
Full	0	1	2	10	170	33490
Oneway	0	1	2	5	15	45
MaxRecDepth	0	1	1	3	21	651

Fig. 2. Impact of control mechanisms on size of test-data sets; 1st row: unconstrained combinatorial coverage (same as G_a in Fig. 1); 2nd row: one-way testing for binary expressions — the resulting numbers reflect that we have eliminated the only source of explosion for this trivial grammar; 3rd row: the recursive depth for operands of unary operators is limited to 1 — explosion is slightly postponed

Now suppose that we face a test scenario such that we need to consider argument sequences of different length, say of length 0, 1 and 2. We may want to further constrain the sequences in an attempt to obtain a smaller data set or simply because we want to honor certain invariants of the grammar-driven functionality under test. Suppose that the order of arguments is irrelevant, and that duplicate arguments are to be avoided. The following annotations express these different approximation intents:

```
[MinLength = 0, MaxLength = 2, NoDuplicates, Unordered] Args ;
```

To enforce a finite data-set, we may impose a depth constraint on expressions:

```
[MaxDepth = 5] Exp ;
```

To summarize, we have illustrated several control mechanisms for combinatorial coverage. These mechanisms require that the test engineer associates approximation intents with sorts, constructors or constructor arguments. The control parameters can be injected into the actual grammar productions, and they can also be given separately.

3 Definition of Combinatorial Coverage

For clarity, we will briefly define combinatorial coverage. (We will use folklore term-algebraic terminology for some formal bits that follow.) Given is a signature Σ and a distinguished root sort, *root* (the latter in the sense of a start symbol of a context-free grammar). As common, we use $\mathcal{T}_\Sigma(\sigma)$ to denote *the set of all ground terms* of sort σ. A *test-data set* for Σ is a subset of $\mathcal{T}_\Sigma(root)$. (We may also consider test-data sets for sorts other than *root*, but a *complete* test datum is of sort *root*.)

We say that $T \subseteq \mathcal{T}_\Sigma(\sigma)$ achieves combinatorial coverage up to depth d for σ if:

$$T \supseteq \{t \mid t \in \mathcal{T}_\Sigma(\sigma), depth(t) \leq d\}$$

Depth of terms is defined as follows; each term is of depth 1 at the least:

$$depth(c) = 1$$
$$depth(c(t_1, \ldots, t_n)) = max(\{depth(t_1), \ldots, depth(t_n)\}) + 1$$

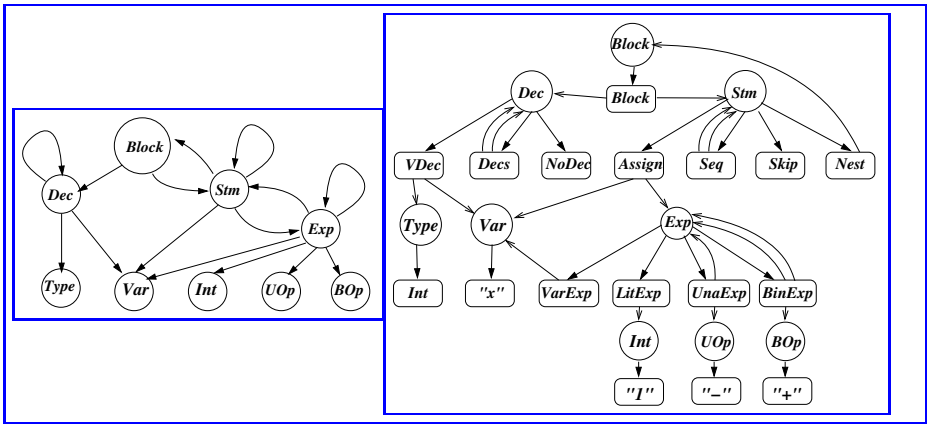

Fig. 3. Sort and constructor graphs for an imperative programming language; there are sorts for program blocks, declarations, statements, expressions, etc. with all the usual constructors; the sort graph is clearly an abstraction of the constructor graph.

It is clear that terms over a signature can be enumerated in increasing depth — the basic algorithm, given below, does just that, in a certain way. *More notation:* We use $\mathcal{T}_\Sigma^d(\sigma)$ to denote the set of all terms of sort σ *at* a given depth d, and we use $\mathcal{T}_\Sigma^{\leq d}(\sigma)$ to denote the set of all terms of sort σ *up to* a given depth d — the latter being the union over all $\mathcal{T}_\Sigma^i(\sigma)$ for $i = 1, \ldots, d$. By definition, $\mathcal{T}_\Sigma^{\leq d}(\sigma)$ is the smallest set that achieves combinatorial coverage up to depth d for sort σ.

4 Grammar Properties Related to Combinatorial Coverage

We will discuss several grammar properties in this section. They are meant to be useful for two purposes: (i) for the *implementation* of test-data generation; (ii) as a *feedback mechanism* for the test engineer who needs to understand the combinatorial complexity of a grammar in the process of modeling the test scenario.

Example. The grammar of expressions in Sec. 2 did not admit any expression terms of depth 1 since all constructors of sort `Exp` have one or more arguments; the minimum depth for expression terms is 2. We call this the *threshold* of a sort. Clearly, one should not attempt to specify a depth limit below the threshold.

All the properties of this section are conveniently described in terms of sort and constructor graphs, which can be derived from any grammar; cf. Fig. 3 for an illustration. The nodes in the *sort graph* are the sorts, while an edge from σ to σ' means that σ' occurs as argument sort of some constructor of sort σ. The *constructor graph* provides a more detailed view with two kinds of

nodes, namely constructors and sorts. There are edges from each sort to all of its constructors. There is one edge for each argument of a constructor — from the constructor node to the node of the argument's sort. Our implementation, *Geno*, compiles the input grammar into an internal representation containing the sort and constructor graph. It uses this graph to direct the generation of objects.

Reachability of sorts from other sorts is extractable from the sort graph. Sort σ' is reachable from σ, denoted by $\rho_\Sigma(\sigma, \sigma')$, if there is a path from σ to σ'. In similarity to terminated context-free grammars, we require that all sorts are reachable from *root*, except perhaps *root* itself. Based on reachability, we can define *recursiveness* of sorts. A sort σ is recursive, denoted by $\mu(\sigma)$, if $\rho_\Sigma(\sigma, \sigma)$. (For a mutually recursive sort, there is a path through other sorts, thereby uncovering a recursive clique. For a directly recursive sort, there is a self-edge.) A stricter form of reachability is *dominance*. Sort σ dominates σ', denoted as $\delta_\Sigma(\sigma, \sigma')$, if all paths from *root* to σ' go through σ. (*root* trivially dominates every sort.) If σ' is reachable from σ, then there is a *distance* between the sorts, denoted as $\varepsilon_\Sigma(\sigma, \sigma')$, which is defined as the shortest path from σ to σ'.

Example. Suppose that the test engineer aims at combinatorial coverage of a specific sort σ up to a given depth d_σ. This implies that the root depth must be at least $d_\sigma + \varepsilon_\Sigma(root, \sigma)$. In case of explosion, the test engineer may review all dominators of σ and limit the recursive depth for them so that the sort of interest, σ, is reached more cheaply.

Using the constructor graph, we can extract the *threshold* of a sort σ, denoted as $\theta_\Sigma(\sigma)$; it is the smallest i such that $\mathcal{T}^i_\Sigma(\sigma) \neq \emptyset$. A more specific threshold can be inquired for each constructor c as denoted by $\theta_\Sigma(c)$. The constructor graph also facilitates shortest term completions both top-down and bottom-up.

5 The Basic Algorithm for Test-Data Generation

There are two overall options for test-data generation: top-down vs. bottom-up. The top-down approach would lend itself to a recursive formulation as follows. Given a depth and a sort, the recursive function for test-data generation constructs all terms of the given sort by applying all possible constructors to all possible combinations of subterms of smaller depths; the latter are obtained through recursive calls.

In Fig. 4, we define an algorithm that adopts the *bottom-up* approach instead. This formulation is only slightly more complex than a top-down recursive formulation, while it offers one benefit. That is, an implementation (using reference semantics for terms) can *immediately* use sharing for the constructed terms; each term construction will be effectively a constant-time operation then (given that the arities of constructors are bounded). It is true that the top-down approach could employ some sort of memoization so that sharing is achieved, too. The bottom-up approach also happens to provide useful feedback to the test engineer. That is, the various sorts are inhabited in increasing depth; so "one can observe explosion", right when it happens.

Arguments
- Signature Σ with root sort, *root*
- Depth $d \geq 1$ for combinatorial coverage

Result Test-data set T that covers Σ up to depth d

Variables
- at^i_σ — terms of sort σ at depth i (i.e., $\mathcal{T}^i_\Sigma(\sigma)$)
- *kids* — an array of sets of terms for building new terms
- *len* — current length of the *kids* array

Notation
- Σ_σ — the set of constructors from Σ that are of sort σ
- $args(c)$ — the sequence of argument sorts for the constructor c
- $kids[1], kids[2], \ldots$ — array subscripting
- $\mathsf{combine}(c, kids, len)$ — build terms with constructor c and subterms from *kids*

Algorithm

```
for i = 1,...,d do begin                    // Term construction in increasing depth
  for each σ in Σ do begin                  // Iterate over all sorts
    at^i_σ := ∅;
    if d − ε_Σ(root, σ) ≥ i then begin      // Skip depth depending on distance from
  root
      if i ≥ θ_Σ(σ) then begin // Skip depth if threshold has not been reached yet
        for each c in Σ_σ do begin          // Iterate over all constructors of sort
          len := 0;
          for each a in args(c) do begin    // Iterate over all arguments of c
            len := len + 1;
            kids[len] := at^1_a ∪ ··· ∪ at^{i−1}_a;   // Determine argument terms
          end;
          at^i_σ := at^i_σ ∪ combine(c, kids, len);   // Build and store terms
        end;
      end;
    end;
  end;
end;
T := at^1_root ∪ ··· ∪ at^d_root;           // Compose result
```

Fig. 4. Basic algorithm for bottom-up test-data generation

We denote the combinatorial construction of terms by $\mathsf{combine}(c, kids, len)$; cf. Fig. 4. Initially, this operation calculates the *Cartesian product* over the term sets for the argument sorts of a constructor (i.e., over *kids*) — modulo a slight detail. That is, a legal combination must involve at least one term of depth $i-1$ (as opposed to $1, \ldots, i-2$); otherwise we were not constructing a term of depth i. Controlled combinatorial coverage caters for options other than the Cartesian product. Hence, $\mathsf{combine}(c, kids, len)$ is subject to redefinition by dependence control; cf. Sec. 6.4.

6 Control Mechanisms for Combinatorial Coverage

We will now define the mechanisms for controlling combinatorial coverage. The basic algorithm, as presented above, will only need simple and local amendments for each mechanism. The following mechanisms will be described:

- Depth control — limit depth of terms; not just for the root sort.
- Recursion control — limit nested applications of recursive constructors.
- Balance control — limit depth variation for argument terms.
- Dependence control — limit combinatorial exhaustion of argument domains.
- Construction control — constrain and enrich term construction.

Several of these mechanisms were illustrated in Sec. 2 complete with additional mechanisms for lists (cf. `MinLength`, `MaxLength`, `Unordered`, `NoDuplicates`). The latter mechanisms will not be formalized here because they are just list-specific instantiations of depth control and dependence control.

6.1 Depth Control

With d as the limit for the depth of terms of the root sort, the *depth limits* for all the other sorts are *implied*. For any given σ, the implied depth limit is $d - \varepsilon_{\Sigma}(root, \sigma)$, and the actual depth may actually vary per occurrence of the sort. This fact suggests a parameterization of the basic algorithm such that a depth limit, d_σ, can be supplied explicitly for each sort σ. The algorithm evolves as follows:

Before refinement

if $d - \varepsilon_{\Sigma}(root, \sigma) \geq i$ then begin // *Skip depth depending on distance from root*

After refinement

if $d_\sigma \geq i$ then begin // *Skip depth depending on sort-specific limit*

All but the depth limit for the root sort are considered optional. (Per notation, d becomes d_{root}.) One should notice that per-sort limits can only *lower* the actual depth limit beyond the limit that is already implied by the *root* limit. More generally, the depth limit for any sort is also constrained by its dominators. Hence, we assume that the explicit depth limits respect the following sanity check:

$$\forall \sigma, \sigma' \in \Sigma.\ \delta_{\Sigma}(\sigma, \sigma') \Rightarrow d_{\sigma'} \leq d_\sigma - \varepsilon_{\Sigma}(\sigma, \sigma')$$

Any control mechanism that works per sort, works *per argument position* of constructors, too. We can view the control-parameter value for a sort as the default for the control parameters for all argument positions of the same sort. Let us generalize control depth in this manner. Hence, we parameterize the algorithm by depth limits, $d_{c,j}$, where c is a constructor and $j = 1, \ldots, arity(c)$. The algorithm evolves as follows:

Before refinement

$$kids[len] := \ at_a^1 \cup \cdots \cup at_a^{i-1}; \qquad \textit{// Determine argument terms}$$

After refinement

$$kids[len] := \ at_a^1 \cup \cdots \cup at_a^{min(i-1,d_{c,len})}; \ \textit{// Determine argument terms}$$

We note that some argument position of a given sort may exercise a given depth, whereas others do not. This is the reason that the above refinement needs to be precise about indexing sets of terms.

6.2 Recursion Control

Depth control allows us to assign *low priority to sorts* in a way that full combinatorial coverage is consistently relaxed for subtrees of these sorts. Recursion control allows us to assign *low property to intermediary sorts* only until combinatorial exploration hits *sorts of interests*. To this end, the *recursive depth* of terms of intermediary sorts can be limited. (For simplicity, we ignore the issues of recursive cliques in the following definition.) The recursive depth of a term t for a given sort σ is denoted as $rdepth_{\Sigma,\sigma}(t)$ and defined as follows:

$$\begin{array}{ll} rdepth_{\Sigma,\sigma}(c) & = \text{if } c \in \Sigma_\sigma \text{ then 1 else 0} \\ rdepth_{\Sigma,\sigma}(c(t_1,\ldots,t_n)) & = \text{if } c \in \Sigma_\sigma \text{ then } 1 + ts \text{ else } ts \\ \quad \text{where } ts & = max(\{rdepth_{\Sigma,\sigma}(t_1),\ldots,rdepth_{\Sigma,\sigma}(t_n)\}) \end{array}$$

Recursion control is enabled by further parameterization of the algorithm. The limits for recursive depth amount to parameters $r_{c,j}$, where c is a constructor and $j = 1,\ldots,arity(c)$. An unspecified limit is seen as ∞. The algorithm evolves as follows:

Before refinement

$$kids[len] := \ at_a^1 \cup \cdots \cup at_a^{min(i-1,d_{c,len})}; \ \textit{// Determine argument terms}$$

After refinement

$$kids[len] := \ \left\{ t \in at_a^1 \cup \cdots \cup at_a^{min(i-1,d_{c,len})} \ \middle| \ rdepth_{\Sigma,a}(t) \leq r_{c,len} \right\};$$

The actual term traversals for the calculation of (recursive) depth can be avoided in an efficient implementation by readily maintaining recursive depth and normal depth as term properties along with term construction.

6.3 Balance Control

Depth and recursion control cannot be used in cases where terms of *ever-increasing depth* are needed (without causing explosion). This scenario is enabled by balance control, which allows us to limit the variation on the depth of argument terms. Normally, when we build terms of depth i, we consider argument terms of depth $1,\ldots,i-1$. An extreme limitation would be to only consider

terms of depth $i-1$. In this case, the constructed terms were balanced — hence, the name: balance control. In this case, it is also easy to see that the number of terms would only grow by a constant factor. Balance control covers the full spectrum of options — with $i-1$ being one extreme and $1, \ldots, i-1$ the other. We further parameterize the algorithm by limits, $b_{c,j} > 1$, where c is a constructor and $j = 1, \ldots, \mathit{arity}(c)$. Again, this parameter is trivially put to work in the algorithm by adapting the step for argument terms (details omitted). An unspecified limit is seen as ∞.

6.4 Dependence Control

We will now explore options for controlling combinations of arguments for forming new terms; recall the discussion of all-way vs. one-way coverage in Sec. 2. The main idea is to specify whether arguments should be varied dependently or independently.

One-Way Coverage. The completely independent exhaustion of argument domains is facilitated by a dedicated coverage criterion, which requires that each argument term appears at least once in a datum for the constructor in question; we say that $T \subseteq \mathcal{T}_\Sigma(\sigma)$ achieves *one-way coverage* of $c : \sigma_1 \times \cdots \times \sigma_n \to \sigma \in \Sigma$ relative to $T_1 \subseteq \mathcal{T}_\Sigma(\sigma_1)$, …, $T_n \subseteq \mathcal{T}_\Sigma(\sigma_n)$ if:

$$\forall i = 1, \ldots, n.\ \forall\ t \in T_i.\ \exists c(t_1, \ldots, t_n) \in T.\ t_i = t$$

We recall that one-way coverage is justified if dependencies between argument positions are not present in the system under test, or they are negligible in the specific scenario. If necessary, we can even further relax one-way coverage such that exhaustion of candidate sets is not required for specific argument positions.

Multi-way Coverage. In between all-way and one-way coverage, there is multi-way coverage reminiscent of multi-way testing (see, e.g., [9]). Classic multi-way testing is intended for testing functionality that involves several arguments. For example, *two-way* testing (or pair-wise testing) assumes that only pair-wise combinations of arguments are to be explored as opposed to all combinations. The justification for limiting combinations in this manner is that functionality tends to branch on the basis of binary conditions that refer to two arguments. In grammar-based testing, we can adopt this justification by relating to the functionality that handles a given constructor by pattern matching or otherwise. For example, some functionality on terms of the form $f(t_1, t_2, t_3)$ might perform parallel traversal on t_1 and t_2 without depending on t_3. Then, it is mandatory to exhaust combinations for t_1 and t_2, while it is acceptable to exhaust t_3 independently. Hence, we face two-way coverage for t_1, t_2 and one-way coverage for t_3.

We further parameterize the algorithm by o_c for each constructor c. The parameters affect the workings of $\mathsf{combine}(c, \mathit{kids}, \mathit{len})$. In turns out that there is a fundamental way of *specifying* combinations. Suppose, c is of arity n. A valid

specification o_c must be a subset of $\mathcal{P}(\{1,\ldots,n\})$. (Here, $\mathcal{P}(\cdot)$ is the power-set constructor.) Each element in o_c enumerates indexes of arguments for which combinations need to be considered. For instance, the aforementioned example of $f(t_1,t_2,t_3)$ with two-way coverage for t_1 and t_2 vs. one-way coverage for t_3 would be specified as $\{\{1,2\},\{3\}\}$. Here are representative specifications for the general case with n arguments, complete with their intended meanings:

1. $\{\{1,\ldots,n\}\}$: all-way coverage.
2. $\{\{1\},\ldots,\{n\}\}$: one-way coverage with exhaustion of all components.
3. $\emptyset$: no exhaustion of any argument required.
4. $\{\{1,2\},\ldots,\{1,n\},\{2,3\},\ldots,\{2,n\},\ldots,\{n-1,n\}\}$: two-way coverage.

This scheme makes sure that all forms of multi-way coverage can be specified. Also, by leaving out certain components in o_c, they will be ignored for the combinatorial exploration. The default for an unspecified parameter o_c is the full Cartesian product. We require minimality of the specifications o_c such that $\forall x,y \in o_c.\ x \not\subset y$. (We can remove x because y provides a stronger requirement for combination.) Options (1.)–(3.) are readily implemented. Computing minimum sets for pair-wise coverage (i.e., option (4.)), or more generally — multi-way coverage — is expensive, but one can employ efficient strategies for near-to-minimum test sets (see, e.g., [26]).

6.5 Construction Control

A general control mechanism is obtained by allowing the test engineer to customize term construction through *conditions and computations*. This mechanism provides expressiveness that is reminiscent of attribute grammars [15]. Thereby, we are able to semantically constrain test-data generation and to complete test data into *test cases* such that additional data is computed by a test oracle and attached to the constructed terms.

We require that conditions and computations are evaluated during bottom-up data generation as opposed to an extra phase. Hence, 'invalid' terms are eliminated early on — before they engage in new combinations and thereby cause explosion. The early evaluation of computations allows conditions to take advantage of the extra attributes. As an aside, we mention that some of the previously described control mechanisms can be *encoded* through construction control. For instance, we could use computations to actually compute depths as attributes to be attached to terms, while term construction would be guarded by a condition that enforced the depth limit for all sorts and constructor arguments. A *native* implementation of depth control is simply more efficient.

We associate conditions and computations to constructors. Given a condition (say, a predicate) p_c for a constructor c, both of arity n, term construction $c(x_1,\ldots,x_n)$ is guarded by $p_c(x_1,\ldots,x_n)$. Given a computation (say, a function) f_c for a constructor $c : \sigma_1 \times \cdots \times \sigma_n \to \sigma_0$ is of the following type:

$$f_c : (\sigma_1 \times A_{\sigma_1}) \times \cdots \times (\sigma_n \times A_{\sigma_n}) \to A_{\sigma_0}$$

Here, A_σ is a domain that describes the attribute type for terms of sort σ. The function observes argument terms and attributes, and computes an attribute value for the newly constructed term. This means that we assume purely synthesized attribute grammars because immediate completion of computations and conditions is thereby enabled. Hence, no expensive closures are constructed, and both conditions and computation may effectively narrow down the combinatorial search space. For brevity, we do not illustrate attributes, but here are some typical examples:

- Expression types in the sense of static typing.
- The evaluation results with regard to some dynamic semantics.
- Complexity measures that are taken into account for combination.

There is one more refinement that increases generality without causing overhead. That is, we may want to customize term construction such that the proposed candidate is replaced by a different term, or by several terms, or it is rejected altogether. This provides us with the following generalized type of a conditional computation which returns a set of attributed terms:

$$f_c : (\sigma_1 \times A_{\sigma_1}) \times \cdots \times (\sigma_n \times A_{\sigma_n}) \to \mathcal{P}(\sigma_0 \times A_{\sigma_0})$$

Geno — our implementation of controllable combinatorial coverage — also provides another form of computations: one may code extra passes over the generated object structures to be part of the serialization process of the in-memory test data to actual test data. Both kinds of computations (attribute grammar-like and serialization-time) are expressed as functions in a .NET language.

7 Testing an Object Serialization Framework

The described grammar-based testing approach has been applied in the mean time to a number of problems, in particular, to differential testing, stress testing and conformance testing of language implementations and virtual processors (virtual machines). *Geno* has been used to generate test-data from problem-specific grammars for Tosca [25], XPath [28], XML Schema [29], the Microsoft Windows Card runtime environment [11], the Web Service Policy Framework [22], and others. Measurements for some applications of *Geno* are shown in Fig. 5.

We will now discuss one grammar-based testing project in more detail. The project is concerned with testing a framework for object serialization, i.e., a framework that supports conversion of in-memory object instances into a form that can be readily transported over the network or stored persistently so that these instances can be de-serialized at some different location in a distributed system, or at some later point in time. The specific technology under test is 'data contracts' as part of Microsoft's WCF. This framework allows one to map classes (CLR types) to XML schemas and to serialize object instances as XML. Data contracts also support some sort of loose coupling.

The overall *testing problem* is to validate the proper declaration of data contracts by CLR types, the proper mapping of CLR types (with valid data contracts) to XML schemas, the proper serialization and de-serialization of object

Status	Grammar	Depth	Time	Terms	Memory
Uncontrolled	WindowsCard	5..	0.05	7.657	1.489.572
	WS Policy	5	1.57	313.041	41.121.608
	Tosca	4	0.08	27.909	2.737.204
	XPath	2	0.09	22.986	2.218.004
Controlled	Tosca	8	0.14	42.210	5.669.616
	Data Contract	6	22.33	2.576.177	365.881.216

Fig. 5. Measuring some applications of *Geno*. Runtime is in seconds, generation time on a Compaq OPTIPLEX GX280, Pentium 4, 3.2 Ghz, 2 Gigabyte of memory. Memory consumption is in bytes. Column 'Terms' lists the number of different terms for the root sort. The *'uncontrolled' measurements* combinatorially exhaust the grammar, except that the length of lists must be in the range 0,1,2. The maximum depth before proper explosion ('out of memory') is shown. In the WindowsCard case, the test set is actual finite; so we write "5.." to mean that test-data generation has converged for depth 5. The depth for Tosca is insufficient to explore expression forms in all possible contexts. The depth for XPath indicates that control is indispensable for generating non-trivial selector expressions. The *'controlled' measurements* take advantage of problem-specific grammar annotations. In the case of Tosca, the corresponding test-data set achieves branch-coverage of a reference implementation. In the case of Data Contract, all essential variation points of the serialization framework are exercised for up to three classes with up to three fields each, complete with the necessary attributes and interface implementations.

instances including round-tripping scenarios. (There are also numerous requirements regarding the expected behavior in case of invalid schemas or CLR types.) Essentially, *Geno* is used in this project to generate classes like the following:

```
[DataContract]
public class Car : IUnknownSerializationData {

  [DataMember]
  public string Model;

  [DataMember]
  public string Maker;

  [DataMember(Name="HP", VersionAdded=2, IsOptional=true)]
  public int Horsepower;

  public virtual UnknownSerializationData UnknownData {
      get { ... } set { ... } // omitted
  }
}
```

In these classes, specific custom attributes are used to inform the serialization framework. The `DataContract` attribute expresses that the class can be serialized. Likewise, fields and properties are tagged for serialization using the

`DataMember` attribute. There is a default mapping from CLR names to XML names, but the name mapping can be customized; see the attribute `Name="HP"`. There are several other attributes and features related to versioning and loose coupling; cf. the implementation of `IUnknownSerializationData` which supports round-tripping of XML data that is not understood by a given CLR type.

The project delivered 7 *Geno* grammars for different validation aspects and different feature sets. The baseline grammar, from which all other grammars are derived by slight extensions, factoring and annotation has 21 nonterminals and 34 productions ("alternatives"). Eventually, these grammars generated about 200.000 well justified test cases. As shown in Fig. 5, *Geno* scales well for grammars of this size. We have also tried to use state-of-the-art test-data generation techniques such as Korat [5], AsmL-Test tool [10] or Unit Meister [27]. However these techniques were not able to cope with the complexity of the serialization problem. (We continue this discussion in the related work section.) The combinatorial search space is due to class hierarchies with multiple classes, classes with multiple fields, various options for custom attributes, different primitive types, potentially relevant interface implementations, etc. The *Geno*-generated test cases uncovered around 25% of all filed bugs for the technology.

8 Related Work

Coverage Criteria for Grammars. Controlled combinatorial coverage is a coverage criterion for grammars, which generalizes on other such criteria. Purdom devised a by-now folklore algorithm to generate a small set of short words from a context-free grammar where each production of the grammar is used in the derivation of at least one word [21], giving rise to *rule coverage* as a coverage criterion. The first author (Lämmel) generalized rule coverage such that all the different occurrences of a nonterminal are distinguished [16] — denoted as *context-dependent rule coverage* (and *context-dependent branch coverage* for EBNF-like expressiveness). Harm and Lämmel defined a simple, formal framework based on *regular path expressions on derivation trees* that can express various grammar-based coverage criteria including rule coverage and context-dependent rule coverage [17]. The same authors also designed a coverage notion for attribute-grammar-like specifications, *two-dimensional approximation coverage*, using recursive depth in the definition of coverage [12]. Controlled combinatorial coverage properly generalizes the aforementioned coverage criteria by integrating depth of derivation, recursive depth of derivation, the dichotomy one-way, two-way, multi-way, all-way as well as the point-wise specification of these controls per sort, per constructor or even per constructor argument.

Grammar-Based Testing. Maurer designed a general grammar-based test-data generator: DGL [19]. The grammar notation is presumably the most advanced in the literature. Productions are associated with weights, but there also features for actions, systematic enumeration, ordered selection of alternatives, and

others. McKeeman described differential testing for compilers and potentially other grammar-driven functionality [20], while test-data generation is accomplished by a 'stochastic grammar'. (Differential testing presumes the availability of multiple implementations whose behavior on a test datum can be compared such that a discrepancy reveals a problem with at least one of the implementations.) Slutz used a similar stochastic approach for differential testing of SQL implementations and databases, even though the grammar knowledge is concealed in the actual generator component [24]. Sirer and Bershad tested Java Virtual machines [23] using 'production grammars' that involve weights as well as guards and actions in order to control the generation process. The weights are actually separated from the grammar so that the grammar may be used in different configurations. This project did not use differential testing but more of a model-based approach. That is, an executable specification of the meaning of the generated JVM byte-code sequences served as an oracle for testing JVM implementations. Claessen and Hughes have delivered a somewhat grammar-based testing approach for Haskell [8], where programmers are encouraged to annotate their functions with properties which are then checked by randomized test data. The approach comprises techniques for the provision of test-data generators for Haskell types including algebraic data types ('signatures'). Again, constructors are associated with probabilistic weights.

Testing Hypotheses. The seminal work on testing hypotheses by Gaudel et al. [4, 2] enables rigorous reasoning about the completeness of test-data sets. Our control mechanisms are well in line with this work. Most importantly, depth control corresponds to a form of a *regularity hypothesis*, which concerns the complexity of data sets. That is, suppose we have a model m and an implementation i, both given as functions of the same type over the same signature, be it $m, i : \mathcal{T}_\Sigma(root) \to r$, where the result type r admits intensional equality. We say that i correctly implements m under the regularity hypothesis for sort *root* and depth d if we assume that the following implication holds:

$$(\forall t \in \mathcal{T}_\Sigma^1(root) \cup \cdots \mathcal{T}_\Sigma^d(root).\ m(t) = i(t)) \implies (\forall t \in \mathcal{T}_\Sigma(root).\ m(t) = i(t))$$

Hence, *any use of a control mechanism for depth or recursive depth for either sorts or constructors or constructors arguments can be viewed as an expression of a regularity hypothesis.* However, our approach does not presume that the complexity measure for regularity is a property of the grammar of even the sort; thereby we are able to express very fine-grained regularity hypotheses, as necessary for practical problems. Dependence control does not map to regularity hypotheses; instead it maps to *independence hypotheses* as common in classic multi-way testing. So our approach integrates common kinds of hypotheses for use in automated grammar-based testing.

Symbolic and Monitored Execution. Our approach does not leverage any sort of existing model or reference implementation for test-data generation. By contrast, approaches based on symbolic execution or execution monitoring support

the derivation of test data from models or implementations. For instance, the Korat framework for Java [5] is capable of generating systematically all non-isomorphic test cases for the arguments of the method under test — given a bound on the size of the input. To this end, Korat uses an advanced backtracking algorithm that monitors the execution of predicates for class invariants, and makes various efforts to prune large portions of the search space. This technique is also embodied in the AsmL-Test tool [10]. Even more advanced approaches use symbolic execution and constraint solving for the purpose of test-data generation [14, 27]. Approaches for execution monitoring and symbolic execution can be very efficient when small intricate data structures need to be generated. An archetypal example of a system under test is a library for AVL trees. These approaches commit to a 'small scope hypothesis' [1, 13], assuming that a high portion of bugs can be found by testing the program for all test inputs within a small scope. (In an OO language, a small scope corresponds to a small number of conglomerating objects.) Hence, these techniques do not scale for the 'large or huge scopes' needed for testing grammar-driven functionality, as discussed in Sec. 7.

9 Concluding Remarks

Summary and Results. Testing language implementations, virtual machines, and other grammar-driven functionality is a complexity challenge. For instance, highly optimized implementations of XPath (the selector language for XML) execute different branches of code depending on selector patterns, the degree of recursion, the use of the reverse axis and the state of the cache. In this context, it is important to automate testing and to enable the exploration of test data along non-trivial complexity metrics such as deep grammar patterns and locally exhaustive combinations. We have described an approach to test-data generation for grammar-based testing of grammar-driven functionality. This approach has been implemented in a tool, *Geno*, and validated in software development projects. The distinguished characteristics of the approach is that test data is generated in a combinatorially exhaustive manner modulo approximations defined by the test engineer. It is indispensable that approximations can be expressed: test engineers can generate test cases that focus on particular problematic areas in language implementations like capacity tests, or the interplay between loading, security permissions and accessibility. We contend that the approach is very powerful, and we have found that test-data generation is unprecedentedly efficient because of the possibility of a backtracking-free bottom-up algorithm that cheaply allows for maximum sharing and semantic constraint checking. Of course, test-data generation is only one ingredient of a reasonable test strategy (others are: grammar development, test oracle, test-run automation), but doing test-data generation systematically and efficiently is beneficial.

Whether or Not to Randomize. Randomized test data generation is well established [3, 7] in testing, in general, and in grammar-based testing, in particular.

The underlying assumption is that the resulting test sets — if large enough — will include all 'interesting cases'. In grammar-based testing, randomized test data generation is indeed prevalent, but we fail to see that this approach would be as clear as ours when it comes to reasoning about testing hypotheses [4, 2], which are crucial in determining appropriateness of test sets. We contend that the weights, which are typically associated with grammar productions, end up fulfilling two *blurred* roles: (i) they specify the relative frequency of an alternative *and* (ii) they control termination of recursive deepening. Instead, controlled combinatorial coverage appeals to hypotheses for regularity and subtree independence by providing designated control mechanisms. Users of *Geno* have expressed that they would like to leverage weights as an *additional* control mechanism, very much in the sense of (i), and we plan to provide this mechanism in the next version. In fact, it is a trivial extension as opposed to the dual marriage: adding systematic test-data generation to a randomized setup is complicated [19, p.54] implementation-wise, and its meaning is not clear either. In our case, weights essentially define filters on subtree combinations.

Future Work. Geno and other tools for grammar-based testing are batch-oriented: the test engineer devises a grammar and test-data generation is initiated in the hope that it terminates (normally). The actual generation may provide little feedback to the test engineer; refinement of generator grammars requires skills and is tedious. We envisage that the expression of testing hypotheses could be done more interactively. To help in this process, a generator should provide feedback such that it reports (say, visualizes) thresholds, distances and explosive cliques in the grammar. (Some ideas have been explored in an experimental extension of *Geno* [30].) A testing framework could also help in taking apart initial test scenarios and then managing the identified smaller scenarios.

Another important area for improvement is the handling of problem-specific identifiers in test-data generation. (Think of variable names in an imperative language.) In fact, this issue very much challenges the generation of statically correct test data. There exist pragmatic techniques in the literature on compiler testing and grammar-based testing; see, e.g., [6, 19, 12]. *Geno* users are currently advised to handle identifiers during test-data serialization in ad-hoc manner. That is, a generator grammar only uses placeholders. Actual identifier generation and the establishment of use-def relationships must be coded in extra strategies that are part of the serialization process. We contend that the overall topic of *general, declarative and efficient identifier handling* deserves further research. For some time in the past, we were hoping that symbolic execution of attribute grammars, as in [12], potentially involving constraint solving, would be a solution to that problem, but its scalability is not acceptable as far as we know of.

Acknowledgments. We are grateful for contributions by Vadim Zaytsev and Joe Zhou. We also acknowledge discussions with Ed Brinksma at an earlier stage of this research. The TestCom 2006 referees have made several helpful proposals.

References

1. A. Andoni, D. Daniliuc, S. Khurshid, , and D. Marinov. Evaluating the "Small Scope Hypothesis". Unpublished; Available at `http://sdg.lcs.mit.edu/publications.html`, Sept. 2002.
2. G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.
3. D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.
4. L. Bouge, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343–360, 1986.
5. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proc. International Symposium on Software testing and analysis*, pages 123–133. ACM Press, 2002.
6. C. Burgess. The Automated Generation of Test Cases for Compilers. *Software Testing, Verification and Reliability*, 4(2):81–99, June 1994.
7. C. J. Burgess and M. Saidi. The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology*, 38(2):111–119, Feb. 1996.
8. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM Press.
9. D. Cohen, S. Dalal, M. Fredman, and G. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–443, July 1997.
10. Foundations of Software Engineering, Microsoft Research. AsmL — Abstract State Machine Language, 2005. `http://research.microsoft.com/fse/AsmL/`.
11. Y. Gurevich and C. Wallace. Specification and Verification of the Windows Card Runtime Environment Using Abstract State Machines. Technical report, Microsoft Research, Feb. 1999. MSR-TR-99-07.
12. J. Harm and R. Lämmel. Two-dimensional Approximation Coverage. *Informatica*, 24(3):355–369, 2000.
13. D. Jackson and C. A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, 1996.
14. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.
15. D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968. Corrections in 5:95-96, 1971.
16. R. Lämmel. Grammar Testing. In H. Hussmann, editor, *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.
17. R. Lämmel and J. Harm. Test case characterisation by regular path expressions. In E. Brinksma and J. Tretmans, editors, *Proc. Formal Approaches to Testing of Software (FATES'01)*, Notes Series NS-01-4, pages 109–124. BRICS, Aug. 2001.
18. Y. Lei and K.-C. Tai. In-parameter-order: A test generation strategy for pairwise testing. In *HASE*, pages 254–261. IEEE Computer Society, 1998.
19. P. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–56, 1990.

20. W. McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.
21. P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–375, 1972.
22. J. Schlimmer et al. Web Services Policy Framework, Sept. 2004. Available at `http://www-128.ibm.com/developerworks/library/specification/ws-polfram/`.
23. E. G. Sirer and B. N. Bershad. Using production grammars in software testing. In USENIX, editor, *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL '99), October 3–5, 1999, Austin, Texas, USA*, pages 1–13, Berkeley, CA, USA, 1999. USENIX.
24. D. Slutz. Massive Stochastic Testing for SQL. Technical Report MSR-TR-98-21, Microsoft Research, Redmond, 1998. A shorter form of the paper appeared in the Proc. of the 24th VLDB Conference, New York, USA, 1998.
25. S. Stepney. *High Integrity Compilation: A Case Study*. Prentice Hall, 1993.
26. K. Tai and Y. Lei. A Test Generation Strategy for Pairwise Testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, 2002.
27. N. Tillmann, W. Schulte, and W. Grieskamp. Parameterized Unit Tests. Technical report, Microsoft Research, 2005. MSR-TR-2005-64; also appeared in FSE/ESEC 2005.
28. W3C. XML Path Language (XPath) Version 1.0, Nov. 1999. `http://www.w3.org/TR/xpath`.
29. W3C. XML Schema, 2000–2003. `http://www.w3.org/XML/Schema`.
30. V. V. Zaytsev. Combinatorial test set generation: Concepts, implementation, case study. Master's thesis, Universiteit Twente, Enschede, The Netherlands, June 2004.