Test-Data Generation for Xtext
Tool Paper

Johannes Härtel, Lukas Härtel, and Ralf Lämmel

Software Languages Team
University of Koblenz-Landau, Germany
http://softlang.wikidot.com/

Abstract. We describe a method and a corresponding tool for grammar-based test-data generation (GBTG). The basic generation principle is to enumerate test data based on grammatical choices. However, generation is broken down into two phases to deal with context-sensitive properties in an efficient and convenient manner. The first phase enumerates test data (i.e., parse trees) with placeholders. The second phase instantiates the placeholders through post-processors. A DSL for grammar transformation is used to customize a given grammar, meant for parsing, to be more suitable for test-data generation. Post-processors are derived from a corresponding object-oriented framework. The actual tool, XTEXTGEN, extends the XTEXT technology for language development.

Keywords: Grammars. Test-data generation. Test-data enumeration. Grammar transformation. Grammar customization. Context sensitivity. XTEXT. XTEND. XTEXTGEN.

1 Introduction

Test-data generation is generally an important method in software engineering and specifically in software *language* engineering; see, e.g., [3,17,18]. In this paper, we are interested in *grammar-based test-data generation* (GBTG) [16,14,8] such that the grammar structure is interpreted for systematic generation of positive and possibly negative examples. Such data can be used to test compilers, interpreters, virtual machines, object serializers, and other language processing components whose input is meant to conform to a given grammar. Scenarios of regression, stress, and identity testing are often addressed in this manner. Based on testing hypotheses for regularity and independence [14], the resulting data sets help revealing issues of language processing components.

In this paper, we advance the field of GBTG.

Contributions of This Paper

- We enhance an existing language modeling technology, XTEXT[1], seamlessly with GBTG. XTEXT readily supports a number of language implementation

[1] http://www.eclipse.org/Xtext/

B. Combemale et al. (Eds.): SLE 2014, LNCS 8706, pp. 342–351, 2014.
© Springer International Publishing Switzerland 2014

aspects related to syntax, e.g., parser generation, model derivation, and error marking, but GBTG was not supported so far.

- We use grammar transformation [5,15] to describe transparently the customization of a grammar meant for parsing to become sufficiently controlled for test-data generation. This approach improves grammar reuse and separation of grammar concerns.
- We treat context-sensitive properties during test-data generation in a systematic manner. To this end, we designate placeholders to the relevant language elements (e.g., identifiers) and instantiate them eventually by post-processors that take a global view on test data (i.e., parse trees).

The paper's website[2] provides access to GBTG resources including Xtextgen.

Road-Map of This Paper. §2 provides an illustrative example in terms of a simple sample language and an associated testing objective. We also discuss relevant challenges in GBTG. §3 describes our GBTG method and the architecture of Xtextgen which implements GBTG for the Xtext technology. §4 describes a transformation-based form of grammar customization, thereby controlling test-data generation. §5 describes a post-processing approach for test data with placeholders so that context-sensitive properties can be handled both efficiently and conveniently. §6 discusses related work. §7 concludes the paper.

2 Illustrative Example

We pick a simple example here: test-data generation for a finite-state machine (FSM) language (FSML). Fig. 1 illustrates FSML with a sample FSM for a turnstile for use in a metro system. Fig. 2 shows a grammar for FSML in Xtext's EBNF-like notation with extra hints at model construction. We want to test language processing components that depend on FSML for their input. Basic examples of such components are an interpreter, a code generator, and a textual-to-visual syntax translation; see [13] for some examples for FSML.

Test-data generation should enumerate fine state machines of increasing complexity while exercising all grammatical choices systematically. One challenge with the basic idea of GBTG is that the combinatorial complexity of the grammar needs to be controlled, e.g., in terms of restricting depth of parse trees or length of lists. Otherwise, the generated sets are simply too large or do not reach syntactical structures of 'interest' before running out of scale. We address this challenge by a designated test-data generation algorithm and grammar transformation-based customization.

Another challenge is that generated test data may need to meet context-sensitive properties because language processing components under test may assume validity with regard to these properties. For instance, FSML readily comes with well-formedness constraints as follows; see [13] for a precise description:

[2] `http://softlang.uni-koblenz.de/xtextgen/`

```
initial state Locked {
  Ticket / Collect -> Unlocked;
}
state Unlocked {
  Pass -> Locked;
  Ticket / Eject -> Unlocked;
}
```

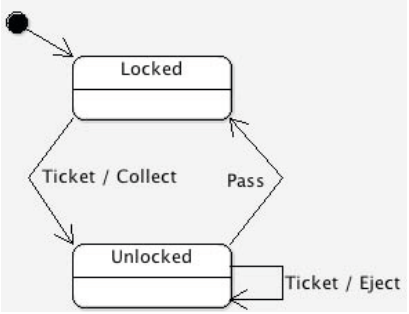

Fig. 1. A finite state machine sample in both textual and visual syntax

```
grammar sle.fsml.FSML with org.eclipse.xtext.common.Terminals

generate fSML "http://www.fsml.sle/FSML"

// A FSM as a collection of multiple states
FSM: states+=FSMState*;

// A possibly initial state with a name and multiple transitions
FSMState:
  (initial?="initial")? 'state' name=ID
  '{' transitions+=FSMTransition* '}';

// A transition with input, optional action and (new) target state
FSMTransition:
  input=ID
  ('/' action=ID)?
  '->' target=[FSMState|ID] ';';
```

Fig. 2. XTEXT grammar of the running example

- There is exactly one initial state.
- The names of all declared states are distinct.
- All states referenced by transitions are also declared.
- The FSM is deterministic.

We handle context-sensitive properties with the help of placeholders for the related parse-tree parts, e.g., identifiers. These placeholders are rewritten to suitable instances in a post-processing phase. A framework of suitable tree-rewriting functions is provided.

3 Method Overview

Consider Fig. 3 for the work and data flow of GBTG according to our method and tool. A test engineer supplies two artifacts: an `.xtext` file, which (semantically, as

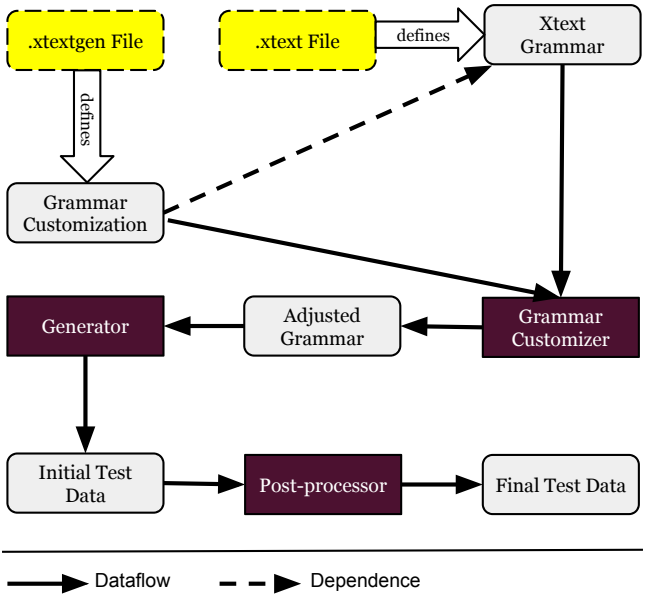

Fig. 3. GBTG with XTEXTGEN

per XTEXT) defines an XTEXT grammar model, and an `.xtextgen` file, which (semantically, as per XTEXTGEN) defines a customization of the grammar at hand. Customization sets up placeholders and limits multiplicities. XTEXTGEN processes the grammar and its customization and returns an 'adjusted grammar' which, in turn, is the foundation for test-data generation. 'Initial test data' may contain placeholders to be instantiated by post-processing to yield 'final test data'. Essentially, XTEXTGEN operates on the XTEXT grammar notation except that arbitrary multiplicities can be expressed and there is a special form $<p>$ to denote placeholder symbols.

The basic generation algorithm *enumerates* test data (parse trees) along the grammatical choices such that we map each grammatical expression to a possibly infinite *sequence*. A case discrimination follows:

- ϵ (epsilon): We use a singleton sequence ['].
- t (terminal): We use a singleton sequence $[t]$
- n (nonterminal): We assume that n is defined in terms of alternatives; see the case for alternatives below.
- $<p>$ (placeholder): Treat as a terminal; see above.
- $x\,y$ (sequence): The juxtapositions of all combinations of elements from x and y are enumerated in a certain order. We use *Cantor pairing*[3] rather than a "nested loop" over the sequences of x and y. In this manner, different elements from the two sequences are more quickly exercised.

[3] `http://en.wikipedia.org/wiki/Pairing_function`

- $x \mid y$ (alternatives): We assume an order of alternatives such that the minimum depth [14] of x is not larger than the one of y. (That it, it is easier to instantiate x than y.) The sequences for x and y are combined by *zipping* them together; x goes first. For instance, [‘1’,‘2’,‘3’,...] and [‘a’,‘b’,‘c’,...] are combined as [‘1’,‘a’,‘2’,‘b’,‘3’,‘c’,...].
- Finite repetitions are mapped as follows:
  - $x? = x^{0,1}$
  - $x^{0,k} = \epsilon \mid x^{1,k}$
  - $x^{1,k} = x \mid x\,x \mid x\,x\,x \mid x \cdots x$ (up to k operands)

There can be infinite sequences indeed, if there is any recursion in the grammar or if there are any infinite repetitions (‘*’ and ‘+’) left past customization. We use one of two strategies in such a case: a) We impose a generic limit on infinite repetitions and recursive depth. b) We only request a finite prefix of some user-specified length, when executing the test-data generator.

The validity constraints in an XTEXT language definition may deal with context-sensitive aspects of the language. The constraints are not generally in a form that they can be used to guide the test-data generation process for valid models. That is, the constraints can be applied to complete parse trees, but they cannot generally be applied to subtrees which arise during generation. It is impractical to filter invalid complete trees afterwards, as too many invalid candidates would be generated. Thus, we generate parse trees with placeholders in a first phase and we apply custom post-processors to instantiate the placeholders. The placeholders deal with identifiers and other syntactic structures that are directly related to the context-sensitive properties.

4 Grammar Customization

In previous work on GBTG [14,9,8], various controls have been investigated, e.g., limits of the depth of parse trees or elimination of combinations according to pairwise testing. Our method uses grammar customization (i.e., transformation) for controlling test-data generation. These transformation operators suffice for the running example:

- “n/i : replace e/k by e'” — In the i-th alternative of nonterminal n, replace the k-th occurrence of grammar symbol (expression) e by e'. If an index (i or k) is omitted, then the first (i.e., the 0−th) alternative or occurrence is assumed.
- “n/i : limit e/k to $b..b'$” — In the i-th alternative of nonterminal n, in the k-th occurrence of e, limit the multiplicity of e to the range $b..b'$. Here, we assume that e is of multiplicity ‘?’ (i.e., 0..1), ’+’ (i.e., 1..∗), or ’*’ (i.e., 0..∗) and $b..b'$ is a proper constraint on the existing lower and upper bound. If ‘$..b'$’ is omitted, then we assume that $b = b'$.

In Fig. 4, we exercise XTEXTGEN’s grammar customization by transformation for the FSML example. In line 1, the customization links to the underlying XTEXT

```
1  customize sle.fsml.FSML
2
3  // Use a more specific name for state names
4  FSMState : replace ID by <state name>;
5
6  // Use more specific names for transition parts
7  FSMTransition :
8    replace ID/0 by <input value>;
9    replace ID/1 by <action value>;
10   replace ID/2 by <state reference>;
11
12 // Require bounds for the number of states
13 FSM : limit FSMState* to 1..6;
14
15 // Limit the number of transitions
16 FSMState : limit FSMTransition* to 1..6;
17
18 // Replace optional "initial" keyword by placeholder
19 FSMState : replace "initial"? by <initial>;
```

Fig. 4. Grammar customization for the running example

grammar. In line 4, we introduce a placeholder *state name* for the occurrence of *ID* in the position of the name of a declared state. In this manner, a post-processor can control the introduction of state names. Likewise, in lines 7-10, we designate specific placeholders to the constituents of a transition, which would otherwise all be generated according to a general notion of *ID*. In line 13, we require that only FSMs with 1 to 6 states are generated. In line 16, we require that the number of transitions per state is between 1 and 6. Here, we assume that we want to limit the combinatorial complexity per state. Finally, in line 19, we replace the optional 'initial' keyword by a mandatory 'initial' placeholder. Thereby, we turn off the combinatorial choice of whether or not to have an 'initial' keyword and we delegate it to post-processing to enforce the constraint of a single initial state.

5 Test-Data Postprocessing

Conceptually, a post-processor is a parse-tree rewriting function. The typical rewrite step is the replacement of a placeholder by a suitable instance. Post-processors may require state, e.g., a custom symbol table, to handle context-sensitive properties. A post-processor may perform branching (by returning multiple output trees per input tree). In principle, a post-processor may also act like a filter (by rejecting input trees). A test-data generator usually combines several post-processors through function composition.

Fig. 5 shows the composition of several post-processors for the running example. Post-processors are programmed in Xtend, which is the Java-like language used with Xtext. The individual post-processors are also described in Xtend

```
// Prepare the individual post-processors
val pickInitial = new PickInitial // Pick an initial state
val removeInitials = new RemoveInitials // Remove remaining placeholders
val nameStates = new NameStates // Assign names to declared states
val useStates = new UseStates // Use valid names in transitions

// Compose the post-processors
val fsmlPP = pickInitial
  .andThen(removeInitials)
  .andThen(nameStates)
  .andThen(useStates)
```

Fig. 5. XTEND post-processors for FSML

```
// A new branch for each match
class PickInitial extends ForEachBranch {
  override protected match(Leaf leaf) {
    return leaf.value == "<initial>"
  }
  override protected build(Leaf leaf) {
    return new Leaf(leaf.label, "initial")
  }
}

// Replace by match by epsilon
class RemoveInitials extends RemoveAll {
  override protected match(Leaf leaf) {
    return leaf.value == "<initial>"
  }
}
```

Fig. 6. Two iterators for treating initial states

while taking advantage of XTEXTGEN's framework of tree-rewriting functions. For instance, Fig. 6 shows the post-processors dealing with the constraint for a single initial state. The first post-processor branches on each possible choice of an initial state and replaces the placeholder by the keyword. The second post-processor removes the placeholders which were not picked in any given branch. In this manner, all options for a single initial state are effectively enumerated.

For brevity, we do not show the XTEND code for the remaining post-processors. Conceptually, *nameStates* generates a new state name for each declared state. The parse tree rewritten by *nameStates* is annotated with the set of generated names so that *useStates* can pick from it by random selection. Thus, both *nameStates* and *useStates* are non-branching (1:1) post-processors. Generally, the ability to pass data between post-processors is an important technique for handling context-sensitive properties.

Test-Data Statistics for the Running Example. To give an idea of the size of test-data sets and execution time of test-data generation we report on two runs of Xtextgen for the FSML example. In the first configuration, we have fully constrained all cardinalities to 1..1. In the second configuration, we have allowed up to 6 states with up to 6 transitions per state; all actions are required.

The measurements were taken on a Windows 8.1 machine with an Intel Core i7-3632QM CPU at 2.20 GHz, 12.0 GB of RAM and a 750 GB harddrive with 8GB of SSD cache. The Java 8 Update 5 runtime environment was used. The generation was executed on Eclipse Luna with DSL developer platform installed, including Xtext 2.6.1. Persistence of test data set was achieved by serializing the parse trees and appending them to a text file using UTF-8 encoding.

Configuration	1	2
# of generated test-cases	1	324726
Size of test-data set	85B	0.257GB
Time for test-data set generation	606.1ms	2403.9s
Time for post-processing	545.6ms (90.0%)	354.0975s (14.7%)
Time for persistence	7.1ms	2049.8s

6 Related Work

Some forms of grammar-based test-data generation have been used in compiler testing for many years; see [3,12] for surveys. In more recent work [16,14,9,8], domain-specific languages for test-data generators have been proposed. These efforts differ in the underlying generation algorithms, the available control mechanisms, (e.g., depth control or pairwise testing) and the linguistic style (e.g., annotation versus custom grammars).

For instance, the YouGen tool [9,8] generates test data by depth while relying on annotations of the nonterminal rules. Annotations control pairwise testing, derivation limits for depth control and Python methods to be applied for global as well as local pre- and post-processing. Xtextgen favors grammar transformation over annotation. Also, placeholders combined with composable post-processors support an effective global view on test data with context-sensitive properties. Xtextgen is fully integrated with Xtext.

The LPTL language [11,10] for test-data specification supports test-driven development with designated IDE support for the language engineer. To this end, the language under test is embedded into the language for test-data specification. Test-data generation is readily mentioned as an excellent complementary approach to LPTL for catching corner cases that the language engineer did not think of.

Our approach is inspired by our previous work [14] in terms of assuming systematic, controlled enumeration of test data. However, there are several important differences. Firstly, we use an enumeration algorithm including Cantor pairing and mandatory multiplicity control as opposed to combinatorial coverage by depth. Secondly, grammars can be reused such that they are customized

by separate transformations. Thirdly, the treatment of context-sensitive properties is more standardized by dedicating an extra phase to placeholder handling on the grounds of a framework of tree-rewriting functions. In previous work, context-sensitive properties were addressed by either complicated formalisms and algorithms limiting scalability of test-data generation [7] or more ad-hoc means of post-processing [14]. Our approach is deeply integrated with Xtext and the corresponding ecosystem; this includes Eclipse.

Xtext bridges between grammarware (text-based concrete syntax) and modelware (EMF-based abstract syntax). This sort of bridging is not completely straightforward [1]. Formalization problems caused by the tree structure of the abstract syntax tree lead to a restriction of the metamodel classes that can be transformed back into the grammar. This relates to the property of (for example) EMF metamodels to provide a containment structure.

In the areas of metamodeling and model transformation, the issue of test-data generation arises, too [2,6]; metamodels are instantiated in a way similar to our approach of using a grammar in generative mode. When testing model transformations, e.g., in model-driven engineering [19,4], test-data generation could be based on both the metamodels of the source models and the transformation description itself. The latter aspect goes beyond our approach and tool.

7 Concluding Remarks

We have described a method and a tool (Xtextgen) for grammar-based test-data generation (GBTG). This effort has been informed by our earlier work on GBTG, specifically [14]. Our main objective is to create a GBTG method and tool that is open-source, well-integrated with an existing technology for language definition (Xtext), suitable for large-scale test-data generation, transparent in terms of achieved grammar coverage, amenable to customization (controls) and handling of context-sensitive properties.

In our experience, practical grammar-based test-data generators tend to treat context-sensitive properties in an ad-hoc manner. In our approach, we aim at leveraging developer knowledge of well-formedness or validity to identify syntactical positions by means of placeholders, which can be instantiated subsequently so that valid test data is obtained.

Several topics remain for future work. We would like to incorporate negative test-data generation into our method. To this end, mutations could be applied systematically to the grammar or to positive test cases directly; see also [20]. We would like to fully enable the level of EMF models as opposed to the Xtextgen-specific parse trees for user interaction with test-data generation, e.g., in the context of post-processing. Finally, we plan to research more deeply on reusing existing validity constraints (as in Xtext's model checkers) for test-data generation. A symbolic execution approach, such as the one used by the Java Pathfinder tool, may help in reusing existing constraints for test-data generation.

References

1. Alanen, M., Porres, I.: A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, Turku Centre for CS (2003)
2. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: An algorithm and a tool. In: ISSRE, pp. 85–94. IEEE (2006)
3. Burgess, C.J.: The Automated Generation of Test Cases for Compilers. Software Testing, Verification and Reliability 4(2), 81–99 (1994)
4. Burgueño, L., Wimmer, M., Troya, J., Vallecillo, A.: TractsTool: Testing Model Transformations based on Contracts. In: Demos/Posters/StudentResearch@MoDELS. CEUR Workshop Proceedings, vol. 1115, pp. 76–80 (2013)
5. Dean, T.R., Cordy, J.R., Malton, A.J., Schneider, K.A.: Grammar Programming in TXL. In: SCAM, p. 93. IEEE (2002)
6. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Software and System Modeling 8(4), 479–500 (2009)
7. Harm, J., Lämmel, R.: Two-dimensional Approximation Coverage. Informatica, 24(3) (2000)
8. Hoffman, D., Ly-Gagnon, D., Strooper, P.A., Wang, H.-Y.: Grammar-based test generation with YouGen. Softw. Pract. Exper. 41(4), 427–447 (2011)
9. Hoffman, D., Wang, H.-Y., Chang, M., Ly-Gagnon, D., Sobotkiewicz, L., Strooper, P.A.: Two case studies in grammar-based test generation. Journal of Systems and Software 83(12), 2369–2378 (2010)
10. Kats, L.C.L., Vermaas, R., Visser, E.: Integrated language definition testing: Enabling test-driven language development. In: OOPSLA, pp. 139–154. ACM (2011)
11. Kats, L.C.L., Vermaas, R., Visser, E.: Testing domain-specific languages. In: OOPSLA Companion, pp. 25–26. ACM (2011)
12. Kossatchev, A.S., Posypkin, M.A.: Survey of Compiler Testing Methods. Programming and Computing Software 31, 10–19 (2005)
13. Lämmel, R.: Another DSL primer, 2013. Technical Documentation. Version 0.00003 as of (December 25, 2013), `https://github.com/slebok/slepro/blob/master/docs/fsml/paper.tex`.
14. Lämmel, R., Schulte, W.: Controllable Combinatorial Coverage in Grammar-Based Testing. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 19–38. Springer, Heidelberg (2006)
15. Lämmel, R., Zaytsev, V.: Recovering grammar relationships for the Java Language Specification. Software Quality Journal 19(2), 333–378 (2011)
16. Maurer, P.: Generating Test Data with Enhanced Context-free Grammars. IEEE Software 7(4), 50–56 (1990)
17. McKeeman, W.M.: Differential Testing for Software. Digital Technical Journal of Digital Equipment Corporation 10(1), 100–107 (1998)
18. Sirer, E.G., Bershad, B.N.: Using Production Grammars in Software Testing. SIGPLAN Notices 35, 1–13 (1999)
19. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal Specification and Testing of Model Transformations. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 399–437. Springer, Heidelberg (2012)
20. Zelenov, S.V., Zelenova, S.: Automated Generation of Positive and Negative Tests for Parsers. In: Grieskamp, W., Weise, C. (eds.) FATES 2005. LNCS, vol. 3997, pp. 187–202. Springer, Heidelberg (2006)