Using Lua features to implement a syntax-based test generator

Cleverton Hentz$^1$ and Anamaria Martins Moreira$^1$

$^1$Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

July 2, 2013
Motivation - Syntax-based Testing

- Grammars are used in many categories of software to describe inputs or some other information. Some examples:
  - compilers and interpreters;
  - software engineering tools;
  - software product line (SPL) configuration descriptions;
  - wireless sensor networks settings;
  - XML (eXtensible Markup Language) specification files.

- Software testing cost can be reduced and its accuracy increased with systematization and automation.
- Coverage criteria limit the number of test cases, keeping a minimum quality and seeking a good set of tests.
- Syntax-based testing is one testing systematization technique.
- It uses the grammar description of the artifact as basis for the definition of the coverage criteria and of the needed test cases.
Motivation - The Lua Language - www.lua.org

- Powerful, fast, lightweight, embeddable scripting language.
- Combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics.
- Dynamically typed and interpreted.
- Ideal for configuration, scripting, and rapid prototyping.
- Has interesting features for the implementation of generators: coroutines and functions as first class objects.

Objectives

- Tool for generating sentences based on the specification of the language;
- Basic infrastructure for the generation of sentences with flexibility in coverage criteria;
- Validation of Lua language features in the implementation of test generators.

and now:
- Expanding the application domain
- Validation and improvement of the tool (flexibility, optimizations, etc.)
Grammar Based Criteria

Classical grammar coverage criteria

- **Terminal Symbol Coverage**: the test suite must contain every terminal symbol of grammar;
- **Production Coverage**: the test suite must contain every production rule of grammar;
- **Derivation Coverage**: the test suite must contain every possible string derivable from grammar.

- LGen implements two coverage criteria: **Terminal Symbol Coverage** and **Production Coverage**;
- Derivation Coverage is attained, when possible, when no other coverage criterion is used.
The Lua Language

- Extension language created by the group TeCGraf of PUC-Rio:
  - Support for several platforms;
  - Dynamically typed;
  - Higher order functions;
  - Tables as the main data structure.

- Functions:
  - Functions in Lua can be anonymous;
  - Any value can be passed back and forth to a function;
  - Multiple values can be returned.

```lua
function g(f)
  return function (a, b) return 2 + f(a, b) end, true
end
```
The Lua Language

- Extension language created by the group TeCGRAF of PUC-Rio:
  - Support for several platforms;
  - Dynamically typed;
  - Higher order functions;
  - Tables as the main data structure.

- Functions:
  - Functions in Lua can be anonymous;
  - Any value can be passed back and forth to a function;
  - Multiple values can be returned.

```lua
function g(f)
  return function (a, b)
    return 2 + f(a, b)
  end,
end
```
Coroutines

- Coroutines characteristics:
  - Own stack;
  - Exclusive instruction pointer;
  - Only one coroutine is performed at a time;
  - User is responsible for controlling the execution.

- Functions for handling coroutines:
  - `create`: creates a new coroutine;
  - `resume`: activates a coroutine;
  - `yield`: suspends a coroutine.

  The function `wrap` creates a coroutine and encapsulates it in a function.

```plaintext
co = coroutine.create(function ()
    print("Hello co", coroutine.yield())
end)

coroutine.resume(co)
coroutine.resume(co) » Hello co

wco = coroutine.wrap(function ()
    print("Hello wco", coroutine.yield())
end)

wco() wco() » Hello wco
```
Coroutines

- Coroutines characteristics:
  - Own stack;
  - Exclusive instruction pointer;
  - Only one coroutine is performed at a time;
  - User is responsible for controlling the execution.

- Functions for handling coroutines:
  - `create`: creates a new coroutine;
  - `resume`: activates a coroutine;
  - `yield`: suspends a coroutine.

- The function `wrap` creates a coroutine and encapsulates it in a function.

```lua
co = coroutine.create(function ()
    print("Hello co", coroutine.yield())
end)
coroutine.resume(co)
coroutine.resume(co) » Hello co

wco = coroutine.wrap(function ()
    print("Hello wco", coroutine.yield())
end)
wco() wco() » Hello wco
```
Coroutines

- Coroutines characteristics:
  - Own stack;
  - Exclusive instruction pointer;
  - Only one coroutine is performed at a time;
  - User is responsible for controlling the execution.

- Functions for handling coroutines:
  - `create`: creates a new coroutine;
  - `resume`: activates a coroutine;
  - `yield`: suspends a coroutine.

- The function `wrap` creates a coroutine and encapsulates it in a function.
Overview of the Tool

- **Input Grammar**: grammar described with EBNF based notation;
- **Lua Specification**: input grammar translated into a Lua table;
- **Selected CoverageCriterion**: the selected coverage criterion used in generation process;
- **Valid Sentences**: set of valid sentences for the specified by input grammar.

**Translator**:
- Implemented with Meta-Environment and Lua PEG;
- Input notation supported: EBNF and ANTLR.

**Generation Engine**:
- Implemented in Lua;
- Provides infrastructure for implementing coverage criteria.
Overview of the Tool

- **Input Grammar**: grammar described with EBNF based notation;
- **Lua Specification**: input grammar translated into a Lua table;
- **Selected Coverage Criterion**: the selected coverage criterion used in generation process;
- **Valid Sentences**: set of valid sentences for the specified by input grammar.

**Translator:**
- Implemented with Meta-Environment and Lua PEG;
- Input notation supported: EBNF and ANTLR.

**Generation Engine:**
- Implemented in Lua;
- Provides infrastructure for implementing coverage criteria.
Overview of the Tool

- **Input Grammar**: grammar described with EBNF based notation;
- **Lua Specification**: input grammar translated into a Lua table;
- **Selected Coverage Criterion**: the selected coverage criterion used in generation process;
- **Valid Sentences**: set of valid sentences for the specified by input grammar.

**Translator**
- Implemented with Meta-Environment and Lua PEG;
- Input notation supported: EBNF and ANTLR.

**Generation Engine**
- Implemented in Lua;
- Provides infrastructure for implementing coverage criteria.
• Component responsible for the recognition and translation of the input notation;
• Initially implemented in ASF+SDF and currently has a version of Lua PEG;
• Input notation could be EBNF based or ANTLR.

**Notation EBNF**

```
Operation = Header_operation, "=" , Level1_substitution;

Level1_substitution = Identity_substitution | Becomes_equal_substitution;
```

**Lua Specification**

```
G. Operation = seq(V.Header_operation, seq(t�rminal("="), V.Level1_substitution))

G. Level1_substitution = +-+-|-(V.Identity_substitution, V.Becomes_equal_substitution)
```
Generation Engine

- Responsible for the generation of sentences and the infrastructure of the coverage criteria;
- Uses coroutines and dynamically created functions to implement the process of generation;
- Input:
  - Lua specification: defines the grammar used in the generation;
  - Derivation control;
  - Coverage criterion configuration.
- Two levels of functions: Grammar Construct Functions (built-in) and Generating Functions (dynamically constructed)
Grammar Construct Functions

- Built-in functions that represent the EBNF constructors;
- Used to define the grammar;
- Return an anonymous standardized function.
Available Grammar Construct Functions

- **Terminal**(str): concatenates the prefix to the terminal str;
- **empty():** represents the empty symbol;
- **Non_terminal**(nt): expands a nonterminal and contains the derivation control.
- **Seq**(patt1, patt2): concatenates the prefix with the results of functions patt1 and patt2;
- **Alt(...):** generates a sentence that begins with the same prefix for each of the arguments;
- **oneOrMoreSep**(patt1, str): repeats the pattern patt1 with separator str.
Generating Functions

- Anonymous functions returned by the grammar construct functions;
- Have a standard signature;
- Return values of type Boolean;
- Their arguments are:
  - \( g \): table that represents the grammar;
  - \( \text{prefix} \): prefix of the sentence being generated by the function;
  - \( \text{maxCycles} \): maximum number of cycles in the derivation of a sentence;
  - \( \text{maxDerivLen} \): maximum number of steps for derivation of a sentence;
  - \( \ldots \): list of nonterminals used in the derivation so far.
Generation Engine Code

```plaintext
function terminal(str)
    return function(g, prefix, level)
        coroutine.yield(prefix..str)
    end
end

function alt(patt1, patt2)
    return function(g, prefix, level)
        patt1(g, prefix, level)
        patt2(g, prefix, level)
    end
end

function seq(patt1, patt2)
    return function(g, prefix, level)
        local m1 = coroutine.wrap(function() patt1(g, prefix, level) end)
        for str in m1 do
            patt2(g, str, level)
        end
    end
end
```
function non_terminal(v)
    return function (g, prefix, level)
        if level > 0 then g[v](g, prefix, level - 1) end
    end
end

function gen(g)
    local m = coroutine.wrap(function () g[g[1]](g, "", 6) end)
    for str in m do
        print(str)
    end
end

g = {
    "nt1",
    nt1 = alt (terminal("a"), seq (terminal("a"), seq (non_terminal("nt1"), terminal("a"))))
}
gen(g)
Derivation Control

- The generation engine provides three mechanisms to control the derivation process independently of any coverage criteria:
  - Restricting the maximum height of a derivation tree \( \text{maxDerivLen} \);
  - Restricting the maximum number of cycles of one nonterminal in a derivation tree branch \( \text{maxCycles} \);
  - Restricting the maximum number of cycles of grammar construct \( \text{oneOrMoreSep} \).
Coverage Criteria

- supports the use of different coverage criteria;
- provides the infrastructure for development and adaptation of new criteria;
- modular, flexible and expandable infrastructure;
- coverage criteria are implemented through an API available in generation engine;
- The functions of this API are divided into two groups:
  - Construction functions calls during the execution of functions of grammar;
  - Control functions calls during the execution of generating functions.
Currently there are two coverage criteria implemented:

- Terminal Coverage;
- Production Coverage.

In both are pre-processed test requirements and execution is controlled to satisfy these requirements;

- **Discard**: checks the generated sentences in relation to satisfaction of criteria.
  - The sentences that do not contribute to satisfy the criteria are discarded.

- **Predictions**: eliminates derivations that do not contribute to satisfying the criterion.
  - *Lookahead* to identify which requirements have been satisfied in the expansion of a nonterminal;
  - Verification of satisfaction of the criteria based on the set of sentences generated until the moment.
Case Study: B Language

- B method is used for formal specification of software components;
- Language used in several projects of group ForAll/Consiste;
- The case study is based on two subsets of the B language:
  - **Small Grammar**:
    - 19 nonterminals;
    - 36 terminals;
    - 26 production rules;
    - Has cycles and the language is infinite.
  - **Expanded Grammar**:
    - 40 nonterminals;
    - 33 terminals;
    - 66 production rules;
    - Has cycles and the language is infinite.
## B Grammar Comparative Results

### Comparison between the results obtained for the small grammar:

<table>
<thead>
<tr>
<th>Exec.</th>
<th>Cycles</th>
<th>MaxDerivLen</th>
<th>Control of Deriv.</th>
<th>Terminal C.</th>
<th>Production C.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>9 - 66.67%</td>
<td>3 - 77.78%</td>
<td>3 - 56%</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>81 - 88.89%</td>
<td>5 - 94.44%</td>
<td>6 - 88%</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>289 - 100%</td>
<td>7 - 100%</td>
<td>8 - 100%</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>289 - 100%</td>
<td>7 - 100%</td>
<td>8 - 100%</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>10</td>
<td>73 - 66.67%</td>
<td>3 - 77.78%</td>
<td>3 - 56%</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>6481 - 88.98%</td>
<td>5 - 94.44%</td>
<td>6 - 88%</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>83233 - 100%</td>
<td>7 - 100%</td>
<td>8 - 100%</td>
<td></td>
</tr>
</tbody>
</table>

### Comparison between the results obtained for the expanded grammar:

<table>
<thead>
<tr>
<th>Exec.</th>
<th>Cycles</th>
<th>MaxDerivLen</th>
<th>Control of Deriv.</th>
<th>Terminal C.</th>
<th>Production C.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>86 - 60%</td>
<td>11 - 69.70%</td>
<td>11 - 54.55%</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>2780 - 87.50%</td>
<td>18 - 90.91%</td>
<td>22 - 86.36%</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>38224 - 97.50%</td>
<td>21 - 96.97%</td>
<td>25 - 96.97%</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>70856 - 97.50%</td>
<td>21 - 96.97%</td>
<td>25 - 96.97%</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>10</td>
<td>4130 - 60%</td>
<td>11 - 69.70%</td>
<td>11 - 54.55%</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>SR</td>
<td>18 - 90.91%</td>
<td>22 - 86.36%</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>SR</td>
<td>21 - 96.97%</td>
<td>SR</td>
<td></td>
</tr>
</tbody>
</table>
Case Study: Software Product Line (SPL) Feature Models

- considered three feature models (available at the SPLOT repository);
- feature models translated into a EBNF grammar;
- characteristics of these grammars:
  - **GPL - Graph Applications**:
    - 16 nonterminals;
    - 13 terminals;
    - 28 production rules;
    - $L(GPL)$ is infinite.
  - **HIS - Home Information System**:
    - 33 nonterminals;
    - 44 terminals;
    - 61 production rules;
    - $L(HIS)$ is finite.
  - **Model Transformation - MD**:
    - 50 nonterminals;
    - 71 terminals;
    - 130 production rules;
    - $L(MD)$ is infinite.
SPL Feature Models Comparative Results

- Terminal symbol and production rule coverage were 100% satisfied;
- Derivation coverage reference for infinite languages - height of 100% coverage of the other criteria.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Criterion</th>
<th>Cycles</th>
<th>MaxDerivLen</th>
<th>Sentences</th>
<th>Run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPL</td>
<td>Derivations</td>
<td>1</td>
<td>8</td>
<td>840</td>
<td>0:00.18</td>
</tr>
<tr>
<td></td>
<td>Productions</td>
<td>1</td>
<td>8</td>
<td>13</td>
<td>0:00.03</td>
</tr>
<tr>
<td></td>
<td>Terminals</td>
<td>0</td>
<td>6</td>
<td>9</td>
<td>0:00.02</td>
</tr>
<tr>
<td>HIS</td>
<td>Derivations</td>
<td>0</td>
<td>8</td>
<td>17920</td>
<td>0:14.16</td>
</tr>
<tr>
<td></td>
<td>Productions</td>
<td>0</td>
<td>8</td>
<td>17</td>
<td>0:00.08</td>
</tr>
<tr>
<td></td>
<td>Terminals</td>
<td>0</td>
<td>8</td>
<td>13</td>
<td>0:00.04</td>
</tr>
<tr>
<td>Model Transformation</td>
<td>Derivations</td>
<td>1</td>
<td>12</td>
<td>1.6391e+26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Productions</td>
<td>1</td>
<td>12</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Terminals</td>
<td>0</td>
<td>10</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

Achievements:

- Satisfaction of the terminal and production coverage criteria for small grammars;
- Significant reduction in the number of sentences generated compared to Derivation criteria;
- Significant reduction of the derivation performed for use of predictions.

Some points to be worked on:

- Satisfaction of the terminal and production criteria for larger grammars;
- Reduction of generation time of the sentences using the coverage criteria.
Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. 