ISOMER — Augmenting Software Testing Confidence by Automated Comparison with a Lightweight Model

Darren Kulp and Daniel Ernst
University of Wisconsin – Eau Claire

Abstract

Non-algorithmic constraints on software development can hinder testing efforts by creating pitfalls for the programmer or by hiding data-dependent errors in complex codes.

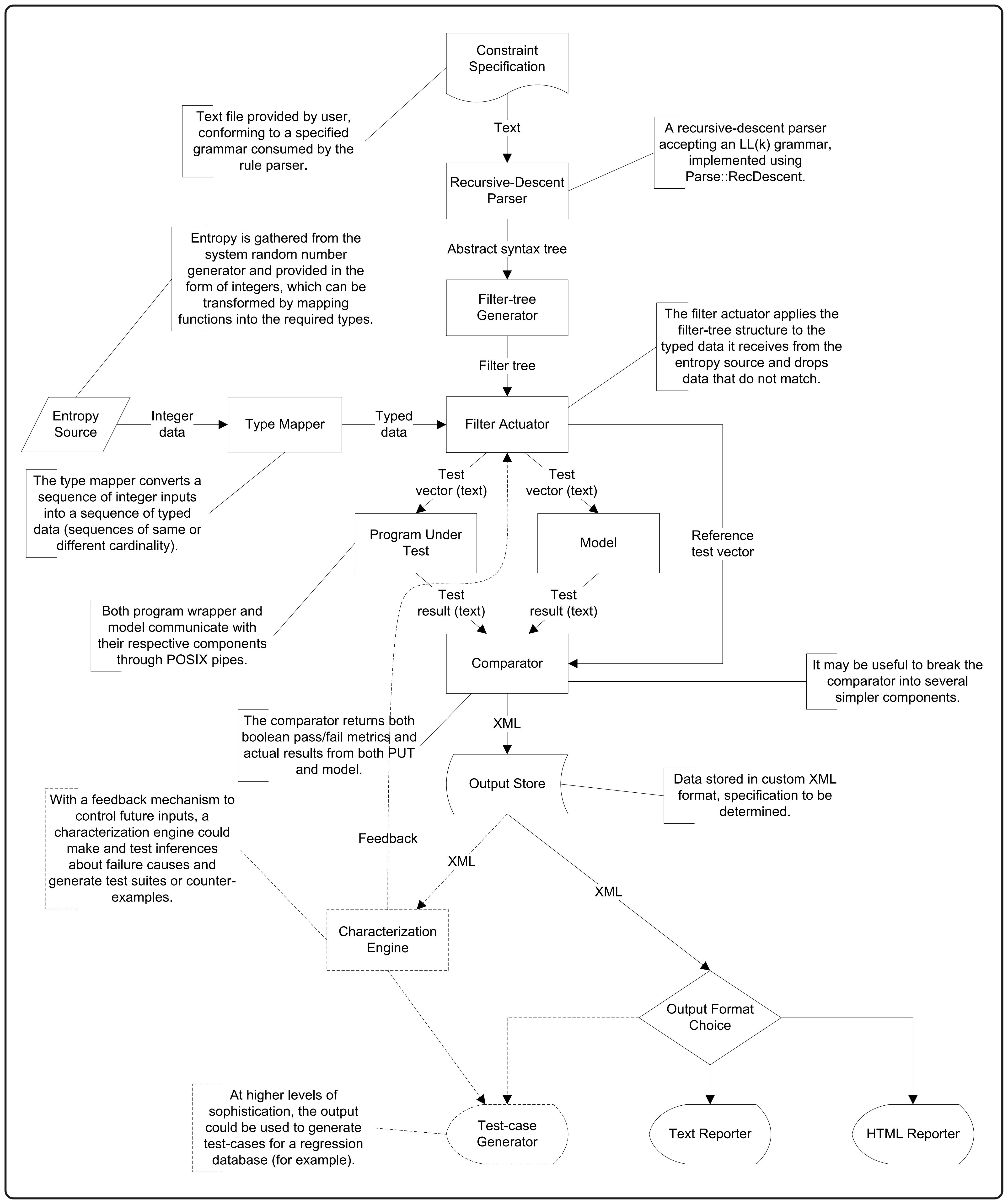
The ISOMER framework improves testing efficacy and confidence by further automating software component and program testing. Using a familiar expression syntax to define constraints on the random stimulus generated, ISOMER subjects software interfaces to dynamically-generated test cases, adding value over time.

Introduction / Design

Creating good test cases is vital to the success of any large software project. Making testing both less tedious and more effective makes finding bugs early more likely and improves software quality and developer productivity over time.

Below is a schematic showing the basic flow of data in the ISOMER system, from user-specified constraints at the top, through automated model-design comparison in the middle, to report generation at the bottom.

The system is self-contained but extensible; the raw data it generates is immediately useful but can be distilled to produce more and different test result data, test cases, bug characterizations, etc.


Constraint Specification
Text file provided by user, conforming to a specified grammar consumed by the rule parser.
Text
Recursive-Descent Parser
A recursive-descent parser accepting an LL(k) grammar, implemented using Parse::RecDescent.
Abstract syntax tree
Entropy is gathered from the system random number generator and provided in the form of integers, which can be transformed by mapping functions into the required types.
Filter-tree Generator
Filter tree
The filter actuator applies the filter-tree structure to the typed data it receives from the entropy source and drops data that do not match.
Entropy Source
Integer data
Type Mapper
Typed data
Filter Actuator
The type mapper converts a sequence of integer inputs into a sequence of typed data (sequences of same or different cardinality).
Test vector (text)
Test vector (text)
Program Under Test
Model
Reference test vector
Both program wrapper and model communicate with their respective components through POSIX pipes.
Test result (text)
Test result (text)
Comparator
It may be useful to break the comparator into several simpler components.
The comparator returns both boolean pass/fail metrics and actual results from both PUT and model.
XML
Output Store
Data stored in custom XML format, specification to be determined.
With a feedback mechanism to control future inputs, a characterization engine could make and test inferences about failure causes and generate test suites or counter-examples.
Feedback
XML
XML
Characterization Engine
Output Format Choice
At higher levels of sophistication, the output could be used to generate test-cases for a regression database (for example).
Test-case Generator
Text Reporter
HTML Reporter

Example

Below is a simple program and configuration to show the use of ISOMER to detect errors in `bc`, the UNIX command-line calculator. It was tested against an intentionally broken version of itself which replaced all occurrences of the "3" on its input with "4".

The constraints definition file demonstrates some possible methods of limiting the random inputs to interesting areas.

The YAML configuration file, which aims for both machine- and human-readability, ties the various components together. Currently specifiable parameters include the shown variables and a seed, which allows the reproduction of a particular input sequence (and, assuming direct mapping from input to output, the same sequence of program and model outputs).

At the left below is the output generated by a sample run of the ISOMER harness on this configuration, showing the outputs trapped as faults. The differences between the model (expected) outputs and design (actual) outputs are highlighted in red. The faults (and in newer versions of the program, all the output segments) are output in YAML for maximum readability while maintaining easy interfacing to output-consumption programs.

```
Seeding with 1035813507 at ./harness.pl line 33.
First failure at test number 2
Stimulus prior to failure:
6 + 1
4 + 29
---
All inputs:
6 + 1
4 + 29
139 + 111
6 + 89
121 + 119
6 + 129
6 + 185
169 + 91
161 + 135
129 + 85
Faults:
---
- dut_output: 260
  input: 139 + 111
  model_output: 250
- dut_output: 306
  input: 161 + 135
  model_output: 296
1..0
```

ISOMER output

```
trials: 10
format: op1 + op2
model: /usr/bin/bc
dut: bc.pl
constraints: bc.txt
```

YAML configuration file

```
byte op1, op2;

constraint ranges {
    op1 < 10 || op1 > 100;
    op1 > 100 -> op2 + op1 > 200;
}

constraint probabilities {
    op1 % 5 inside [ 1, 4 ];
    op2 % 10 != 3;
}

constraint parity {
    op1 % 2 == op1 > 100;
    op2 % (3 - 1 * 2 + 1) == 1;
}
```

constraints definition file

Conclusion

We believe the underlying concept behind constrained-random-stimulus testing to be solidly proven in theory and in real use with hardware, and we feel an exploration of its application to software development will result in long-term improvements to developer productivity and product quality.

ISOMER has explored this application in a language- and platform-independent manner, distinguishing itself from other systems in the magnitude of possible productivity gains. In proving the potential of a cross-platform, cross-environment, specification-based testing framework, ISOMER reveals previously unharnessed gains in tester and developer productivity.