

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

Darren Kulp and Daniel Ernst  
Department of Computer Science  
University of Wisconsin – Eau Claire  
Eau Claire, WI 54702  
{kulpdm,ernstdj}@uwec.edu

## **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.

# 1 Introduction

The development of any complex piece of software inevitably uncovers numerous faults. Unfortunately, not all bugs are caught in the time (if any) set aside for debugging; every bug that escapes to production reduces the value of the finished product and increases total costs. Bugs caught early are easier to repair than bugs caught late. Some types of bugs, particularly data-dependent bugs, resist detection, surfacing only in relatively rare cases which may never occur during normal testing.

Our system “ISOMER” helps find data-dependent errors in software programs or components by subjecting their input interfaces to random inputs constrained to thoroughly exercise their input space, and by comparing their output with that generated by lightweight models. Because the input is generated randomly but constrained to valid and “interesting” areas by programmer-supplied rules, the testing framework that results adds value over time; unlike normal “directed” unit tests, which test fixed input and output combinations, our testing system continually produces new tests and improves confidence in correctness and robustness.

In section 2, we will overview previous work and background information that bears on our design and implementation. Section 3 will describe the novel aspects of our design; section 4 will discuss implementation details thereof. Section 5 evaluates the goals of the system in spheres of functionality and performance, and section 6 concludes this document.

## 2 Background

### 2.1 Problem

In some development environments, there can be a significant expenditure of time by the programmer on issues stemming not from the problem to be solved but from constraints imposed by the environment. Heap management, for example, can be tedious and error-prone, but is an integral part of practically every production C / C++ program. Typical test cases may not account for unexpected behavior stemming from such errors, and even if a developer is aware of the possibility of such faults, it is difficult or impossible to design a static test or suite that will detect them. Of course, problems caused by heap management are only a possible manifestation of the more general phenomenon: data-dependent failures.

## 2.2 Use of random stimulus

Random stimulus as a basis for testing has precedent in both hardware design and software development; we mention hardware design specifically because it was the original inspiration for this project [2] and because its use in hardware is more mature and widespread.

### 2.2.1 In hardware design verification

ISOMER's premise of constrained-random inputs and lightweight models is not novel; it exists today in the domain of hardware design verification [2]. A lightweight model of the device or design under test ("DUT") is constructed, to which random inputs, constrained by declarative code to valid and/or "interesting" areas, are fed. The same stimuli are supplied to the real design, and the outputs are compared. Assuming the model perfectly implements the design's designed behavior and that the comparison code is perfect, a mismatch indicates an error in the DUT. The efficacy of the concept lies in the word "lightweight": the model is necessarily easier to construct than the DUT.

### 2.2.2 In software testing

This system is well suited to hardware testing, partly due to the stable, synchronous interfaces of hardware components, and partly due to the great amount of simplification that can be achieved by a model over the real implementation. In hardware design, a significant, sometimes enormous amount of effort is expended not on algorithmic complexity, but on time or space optimizations, since hardware's realization in silicon constrains it in ways unknown to software.

These techniques have not been as commonly applied to software, but solutions do exist under the name of "automatic specification-based testing" [3] [6]. See section 2.3 for more specific information on software precedents.

External constraints in hardware or software design can create their own problems, sometimes the most insidious and difficult to diagnose. In hardware design, external constraints include timing problems and physical implementation size; in software design, they include implementation language and library requirements. For example, when using C or C++, where it is necessary for the programmer to manage memory allocation and where failure to do so properly can result in hard program failure, it can be difficult to debug data-dependent errors, and it is especially vital to ensure that such bugs do not remain hidden, since error recovery at run-time is often poorly supported – segmentation faults are not user-friendly.

In such situations, a lightweight model implemented in a rapid-development language such as Perl or Python can be developed in parallel with its heavyweight equivalent to test correctness. This lightweight model might even act as a prototype, implemented before

the heavyweight version; in cases where this prototype stage already exists, adding the constrained-random-stimulus system represents a significant gain – greatly improved confidence in program correctness – at low cost – small amounts of glue code to harness the test system to the implementations.

## 2.3 Existing tools in the domain

As mentioned previously, tools attacking the problem addressed by ISOMER exist, but generally suffer from fundamental limitations including

- dependence on a particular language or platform
- restriction to synchronous function interfaces

The original inspiration for the ISOMER project was the Synopsys Verification Methodology Manual for SystemVerilog [2], which defines a technology and a methodology for subjecting hardware designs in hardware description languages like Verilog to interesting randomized inputs. In this system, the “model” we refer to is called a “scoreboard”, and includes the logic used to compare actual outputs with expected ones; in our design, these functions are separated.

In the software universe, this technique, automatic specification-based testing, is most preceded in functional environments. The code on which ISOMER is based, LectroTest [6], is itself written in Perl, which is not a specifically functional language, but it is based on and borrows heavily from an implementation called QuickCheck [3], whose website links to implementations in Erlang, Scheme, Common Lisp, Python, and ML, all but one of which are functional languages. ISOMER therefore explores relatively uncharted territory in applying these techniques to alternate programming models.

## 3 Novel design

### 3.1 Goal

Our project facilitates the detection of data-dependent errors by, among other things,

1. more thoroughly exploring the problem input space than otherwise feasible
2. collecting counterexamples / problematic inputs for regression tests
3. providing an extensible framework on which to build additional tools

### 3.1.1 Input space exploration

The fundamental advantage of constrained-random-stimulus (CRS) testing is that the tests it creates are both dynamic and intelligently guided to interesting areas. It is here that it differs from existing testing tools and methods including unit testing (“*xUnit*” for various values of  $x$ ) and fuzzing tools (for example, *fuzz* and *zzuf* [5]). CRS is more dynamic than *xUnit*, and more intelligent than fuzzers. ISOMER, while having important similarities to these tools, does not replace them.

The fundamental difference between ISOMER and a unit-testing system is the presence in ISOMER of a lightweight model and associated input-generating code. Typical unit tests are “directed,” meaning they check for specific, pre-recorded responses generated in response to specific, pre-recorded inputs. ISOMER’s use of a model and uniform, entropic inputs subject to programmer-specified rules potentially greatly increases confidence in program correctness because of the relative *dynamicity* of the tests produced by ISOMER.

ISOMER also differs from “fuzzing” programs, which attempt to produce program crashes by subjecting the program to random input. Fuzzing, however, does not generally produce valid stimuli and does not make use of a reference model, so it can increase confidence only in program robustness in the face of erratic input and not in program correctness. An implementation of our method with no specified constraints (or with constraints that do not represent the valid program input space) is functionally equivalent to a fuzzing program, but such a use does not involve behavior modeling and is therefore unrelated to the project goal. In addition, ISOMER, unlike fuzzers, is not limited to interacting with user-level interfaces (stdin/stdout, for example), but can access APIs directly as well.

### 3.1.2 Counterexample generation

The output produced by ISOMER is of various types; it can replay the exact inputs needed to produce a particular failure, it can be adapted to turn such a replay into a static, “directed” (possibly *xUnit*-style) test, or it can simply collect the random seeds that deterministically led it to a set of failures, which can be regenerated later. The ability of the system to “replay” a failure scenario (to generate a “counterexample” to a property assumed about program behavior) is very useful both for integration with existing test systems and for regression testing.

### 3.1.3 Lightweight modeling

It is worth discussing the meaning of “lightweight” in the context of software development. In hardware design (the original inspiration for this technique), external constraints such as timing and space requirements and concerns about simulating physical characteristics of a design add greatly to the design burden. These non-algorithmic concerns are stripped away

in a lightweight hardware model, resulting in an algorithmically-equivalent but significantly simpler-to-develop design. In software, similar external constraints (e.g., implementation language, frameworks, human-interaction interfaces) exist, but they are often less separable and less obvious than their hardware counterparts. It is well-recognized that some languages and development environments are well-suited to Rapid Application Development (RAD), and it is still true that a large portion of commercial software is not developed with these tools, due in many cases to their immaturity or other stigma. In ISOMER’s context, “lightweight” can mean “developed using a RAD tool or methodology instead of one imposed by production constraints.” The efficacy of our system depends on the rapidity of development and ease of maintaining the model; a C++ model for a C++ product, for example, is very unlikely to represent an overall gain, unless there are significant algorithmic optimizations that can be avoided in model development.

## **3.2 Limitations**

### **3.2.1 Conceptual limitations**

Despite the advantages to automatic specification-based testing (constrained-random verification), there are important limitations. Perhaps most obvious is the fact that a model must be written; although in some cases the model can be harvested from a prototype created before the production version, more often it must be written from scratch. If the advantages of ISOMER are not clear, a developer or team might be convinced with difficulty to add this to an already-compressed workload. We believe that the long-run advantages of ISOMER far outweigh the initial investment time, but this is a decision that must be made per project.

It is important to note that the constrained-randomly-generated-stimulus approach is not universally applicable; some problem domains do not lend themselves to “lightweight” modeling. For example, constraining random stimuli into a valid MPEG stream may not be reasonably implemented in a lightweight fashion. However, there remains a large class of algorithmic problems to which this approach could be usefully applied.

### **3.2.2 Technical limitations**

ISOMER aims to be a more general solution to the problem presented than the existing tools; by using a language-neutral syntax (unlike the mixed Haskell/Perl syntax of LectroTest [6]) and by using technology-independent, time-proven interfaces like POSIX pipes, ISOMER offers a single system that can be used across multiple projects and which can be learned without previous knowledge of a particular platform. However, this flexibility comes at a cost: a POSIX-pipes-only implementation makes interacting with whole programs easier but increases the effort necessary to interact with more granular components like functions. Language-specific shims can be automatically generated to limit the

impact of this limitation, but clearly an interface cannot be simultaneously entirely generic and perfectly granular.

### 3.3 Usage examples

For example, take a solution engine for the popular Sudoku numbers puzzle, written in C. Such a system might well use a backtracking engine with state objects being created and destroyed thousands or millions of times during a single program invocation. A programmer used to C++ or Java and their use of implicit copy constructors or garbage-collected references, respectively, might mistakenly return a pointer to a state stored on the program stack, for example. If this bug is hidden in an infrequently-used code path that is not well-exercised in the developer's test cases, it may go unnoticed for a long time. Even a simple set of constraints fed to ISOMER and allowed to run nights and weekends on the developer's own machine significantly raise the likelihood of discovering such a flaw by producing a set of inputs that produce deviant output. More carefully-written constraints and dedicated compute nodes can attack even far more complex problems. Another example might comprise a bio-informatics program developed in C++ compared to a rapidly-developed version using Perl and existing tools like BioPerl [1]. In cases like these, where the solution being developed is an innovation on an available tool or process rather than an entirely new one, the cost of developing the model may be negligible: BioPerl programs can't compete with commercial variants in speed, but they may well compete in correctness, which is the only requirement ISOMER makes.

Another field where ISOMER could be useful is the verification of multi-threaded applications when compared to their single-threaded ancestors or to a simpler, single-threaded but equivalent model. With the recent advances in multi-core processor technology, multi-threaded software is becoming necessary more rapidly than it is becoming available. Using ISOMER to compare a new multi-threaded system with a known-good single-threaded version or prototype could capture race conditions or undefined behavior much more quickly than a static testing framework could.

## 4 Implementation

Our implementation so far comprises the following elements:

1. a grammar (Figures 1 and 2) for the constraint system
2. a parser which consumes that grammar and generates an abstract syntax tree
3. a reserialization component (for testing and programmatic modification)
4. a random stimulus generator, producing reproducible sequences of typed data

```

1 <autotree>
2 Top:      Declaration(s?) Constraint(s?) EOF { bless +{
3           declarations => bless($item[1], 'DeclarationList'),
4           constraints => bless($item[2], 'ConstraintList')
5         }, $item[0] }
6 Declaration: Type Dimension(s?) Ident(s /,/) ';' {
7           $thisparser->Extend("VarName: "
8             join ' | ', map '/\b'. $_->_dump.'\b/', @{$item[3]});
9           bless +{ type => $item[Type],
10                  dimensions => $item[2],
11                  idents => $item[3] }, $item[0];
12         }
13 Type:     'int' | 'nybble' | 'byte'
14 Dimension: '[' /[0-9]+/ ']'
15 Constraint: 'constraint' Ident CGroup
16 Evaluable: VarName { $item[1] } | Number { $item[1] }
17 VarName:  <reject> # Starts out empty, gets extended by Declaration
18 Ident:    /\b[A-Z_]\w*\b/i
19 Number:   /\b\d+\b/
20 CGroup:   '{' Statement(s /;/) ';' '}' { bless $item[2], $item[0] }
21 Implication { $item[1] } | BExpr { $item[1] }
22 BExpr:    InsideExpr { $item[1] } | CGroup { $item[1] } | IExpr { $item[1] }
23 IExpr:    ISEExpr[$arg[0] || 0] { $item[1] } |
24           IUExpr[$arg[0] || 0] { $item[1] } |
25           IPEExpr { $item[1] }
26 IPEExpr:  '(' IExpr[0] ')'
27 ISEExpr:  <leftop:IAtom[$arg[0]] IBiOp[$arg[0] || 0] IAtom[$arg[0]]> {
28           if (@{ $item[1] } == 1) { $item[1][0] }
29           else { bless $item[1], $item[0] }
30         }
31 IUExpr:   IUnOp IAtom[$arg[0]]
32 IBiOp:    <matchrule:IBiOp$arg[0]>
33 IBiOp8:   <commit> <reject>
34 IBiOp7:   '***'
35 IBiOp6:   '*' | '/' | '%'
36 IBiOp5:   '+' | '-'
37 IBiOp4:   '<<' | '>>'
38 IBiOp3:   '&' | '|' | '^'
39 IBiOp2:   '<=' | '>=' | '<' | '>'
40 IBiOp1:   '==' | '!='
41 IBiOp0:   '&&' | '||'
42 IUnOp:   '~' | '-' | '+'
43 IUnOp:   '!'
44 IAtom:   IUExpr[$arg[0] || 0]
45           { ($arg[0] || 0) < 8 | undef } IExpr[($arg[0] || 0) + 1] { $item[2] } |
46           Evaluable { $item[1] }
47 InsideExpr: IExpr 'inside' '[' IList ']'
48 IList:      IAtom(s /,/) { bless $item[1], $item[0] }
49 Implication: BExpr '->' BExpr { bless [ @item[1,3] ], $item[0] }
50 EOF:       /\z/

```

Figure 1: Grammar text

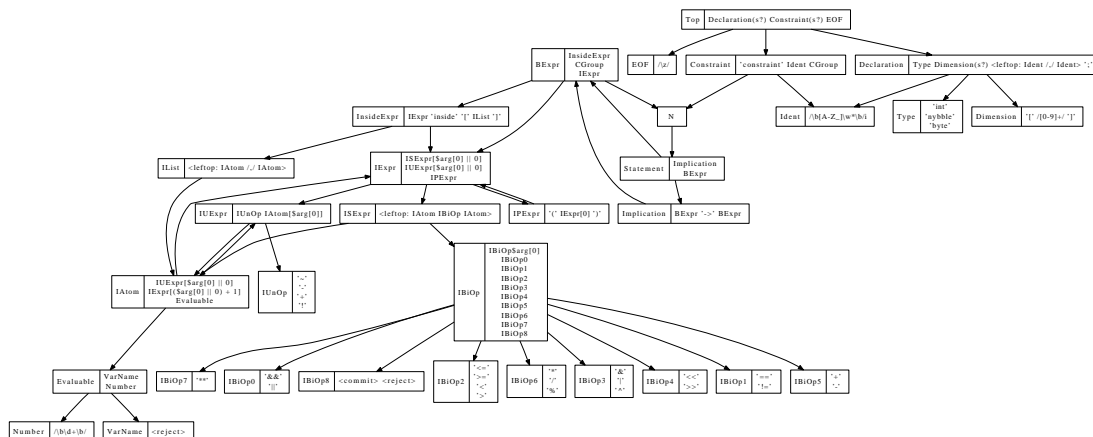


Figure 2: Grammar (graphical depiction)

```

1  int a, b, c, d, e;
2  int[100] large_array, intersecting_array;
3
4  constraint shell_game {
5      l inside [ a, b, c ];
6      a == 1 -> b == c && c == 0;
7      b == 1 -> c == a && a == 0;
8      c == 1 -> a == b && b == 0;
9  }
10
11 constraint complex_constraint {
12     a >= 0 -> {
13         c == 0 || b inside [ 1, 2, 3, 5 ];
14         a inside [ 1, 2, large_array ] -> !a inside [ intersecting_array ];
15     };
16     a < 0 -> ~(b >> 2) & c ^ d == e;
17 }

```

Figure 3: Example constraint file

5. a DUT wrapper, providing a common input/output interface to DUTs using POSIX pipes
6. an output comparator

The grammar is semantically complete but may eventually be extended to differentiate integer and Boolean expressions, which are currently treated in a weakly-typed fashion, as in Perl or C.

The grammar and parser are implemented using the `Parse::RecDescent` parsing module [4] for Perl. This parser-building framework builds, as its name implies, a recursive-descent parser from an LL(k) grammar. The potential inflexibilities of an LL(k) grammar were outweighed by the ease of producing and understanding such a grammar.

An example of a file that conforms to the grammar can be found in Figure 3. This arbitrary set of constraints does not represent the requirements of a typical constraint problem, but it does demonstrate the expressiveness and readability of the grammar. Developers familiar with C-like languages will feel at home with the syntax; the mathematical operations follow the semantics and precedence of the host language (i.e., Perl, and therefore very C-like).

A schematic diagram showing ISOMER’s main components is shown in Figure 4. The top two boxes represent the combination of the grammar in Figure 1 and the parser; the third from the top is a translation layer that converts the expression tree into a format compatible with `Test::LectroTest`. The “Filter Actuator” is the `Test::LectroTest` framework’s `TestRunner`, being supplied entropy on the “left” by its generation framework. The arrows emanating from the actuator represent the parallel POSIX pipes to the concurrently-running model and DUT (“PUT” in the diagram); the comparator is, due to the simple line-oriented interface, simply a patterned string comparison.

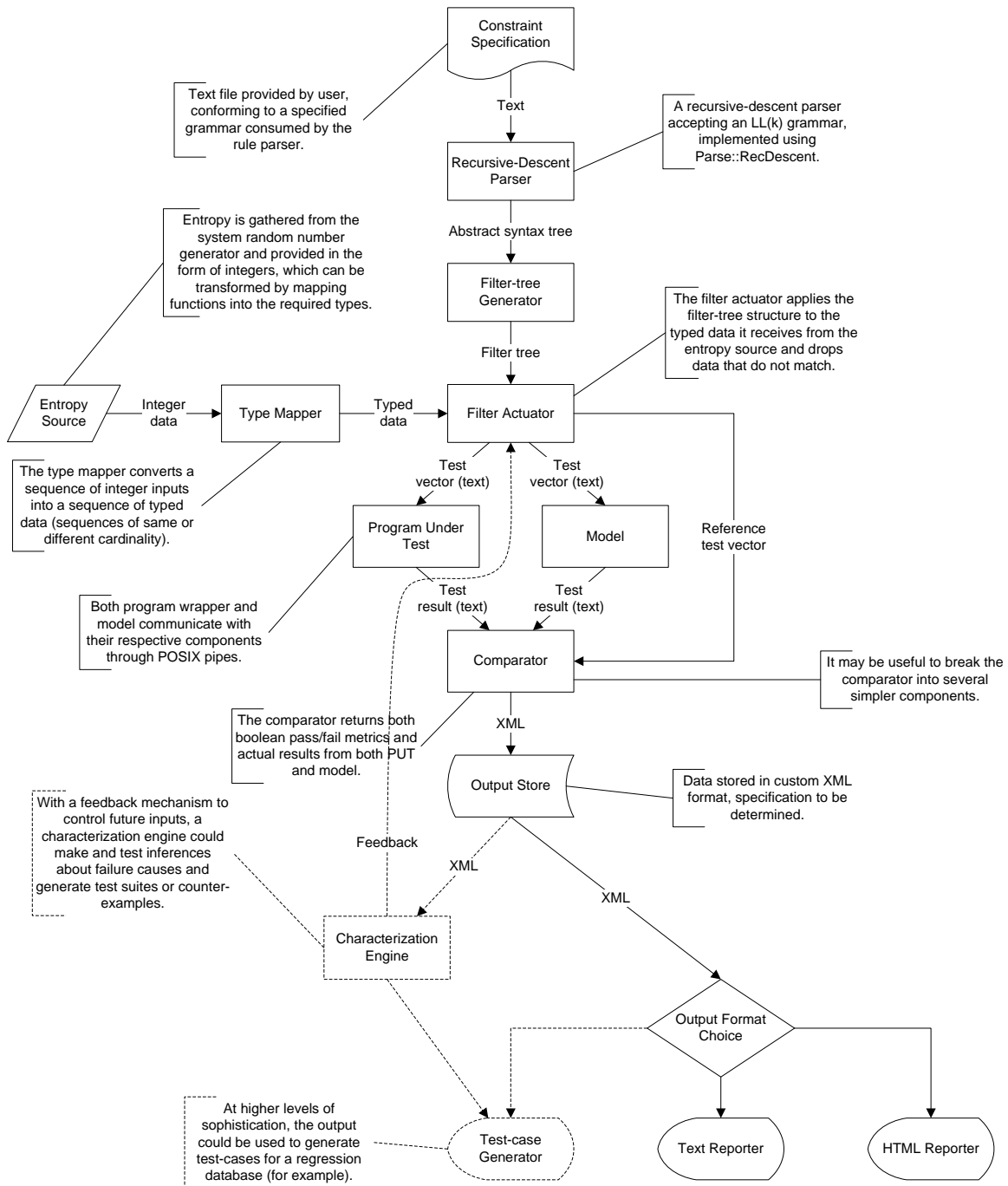


Figure 4: Schematic diagram

## 5 Evaluation

Two important areas in which to evaluate ISOMER are its level of functionality and its performance constraints. We address these in the paragraphs following.

### 5.1 Functionality

The functional evaluation can be divided into two subsets: type support and interface or I/O support. Other functional areas of course exist; these are simply the two spheres where ISOMER differentiates itself most from its predecessors.

#### 5.1.1 Type support

ISOMER currently supports a limited set of types, specifically 32-bit, 8-bit, and 4-bit signed integers. Support in the underlying LectroTest framework exists for more complex types, including strings, lists of other types, and arbitrary structural types through concatenation. In order to support some of these types, ISOMER's grammar will have to be extended; for the present we limit ourselves to integral types for simplicity.

#### 5.1.2 I/O support

As mentioned previously, ISOMER communicates with programs using POSIX pipes. Even under the current system, access to library functions and other program-internal components can be effected using shims. These shims must currently be created individually, but their creation can be automated in the future on a per-language basis to provide easy access to the function-level granularity that other CRS systems offer.

#### 5.1.3 Functionality example

To demonstrate ISOMER's functionality simply, we set up a constraint configuration with a purposely broken version of *bc*, a UNIX command-line calculator. The broken version substitutes the digit 4 for the digit 3 in its input, causing most expressions containing a 3 to fail. The "model" in this case was the normal version of *bc*. Figure 5 shows the log of a brief (5-trial) run of ISOMER on this configuration.

```

1 Seeding with 686968504 at ./harness.pl line 33.
2 First failure at test number 2
3 Stimulus prior to failure:
4 2 + 1
5 1 + 1
6 2 + 1
7 1 + 1
8 2 + 3
9 3 + 1
10 4 + 2
11 Faults:
12 ---
13 - dut_output: 6
14   input: 2 + 3
15   model_output: 5
16 - dut_output: 5
17   input: 3 + 1
18   model_output: 4
19 1..0

```

Figure 5: Example ISOMER run on *bc*

## 5.2 Performance

It was foreseen that ISOMER might create significant run-time overhead, which, if sufficiently severe, could render large-scale use impracticable. The current implementation has not been subjected to significant optimizations; as Donald Knuth said, “Premature optimization is the root of all evil.” At the moment, the filter-based entropy system, which does not benefit from any compilation of constraints to optimized generators, consumes most of the time in a typical run. While a simple design under test (a doubly-linked-list library wrapper) consumed 250 tests in 11ms, the entire run took on average about 9400ms to complete.

Clearly this level of performance is undesirable, but it is important to note that this major gap in performance closes as the DUT slows. For a non-trivial DUT, the time per test might be orders of magnitude slower, while the constraint generation time per test, which is stable and dependent directly on the complexity of the constraints (only negligibly on the number of variables being constrained), and since the generation and consumption of tests occur concurrently, the loss is not so great as it first appears. Naturally, performance issues will be explicitly addressed in future revisions.

## 6 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.

## References

- [1] Main Page - BioPerl. <http://www.bioperl.org/>.
- [2] BERGERON, J., CERNY, E., HUNTER, A., AND NIGHTINGALE, A. *Verification Methodology Manual for SystemVerilog*. Springer, 2005.
- [3] CLAESSEN, K., AND HUGHES, J. QuickCheck: An Automatic Testing Tool for Haskell. <http://www.cs.chalmers.se/~rjmh/QuickCheck/>.
- [4] CONWAY, D. Parse-RecDescent-v1.95.1 - search.cpan.org. <http://search.cpan.org/dist/Parse-RecDescent/>.
- [5] HOCEVAR, S. zzuf - multi-purpose fuzzer. <http://sam.zoy.org/zzuf/>.
- [6] MOERTEL, T. Moertel Consulting's Community Projects :: LectroTest. <http://community.moertel.com/ss/space/LectroTest>.