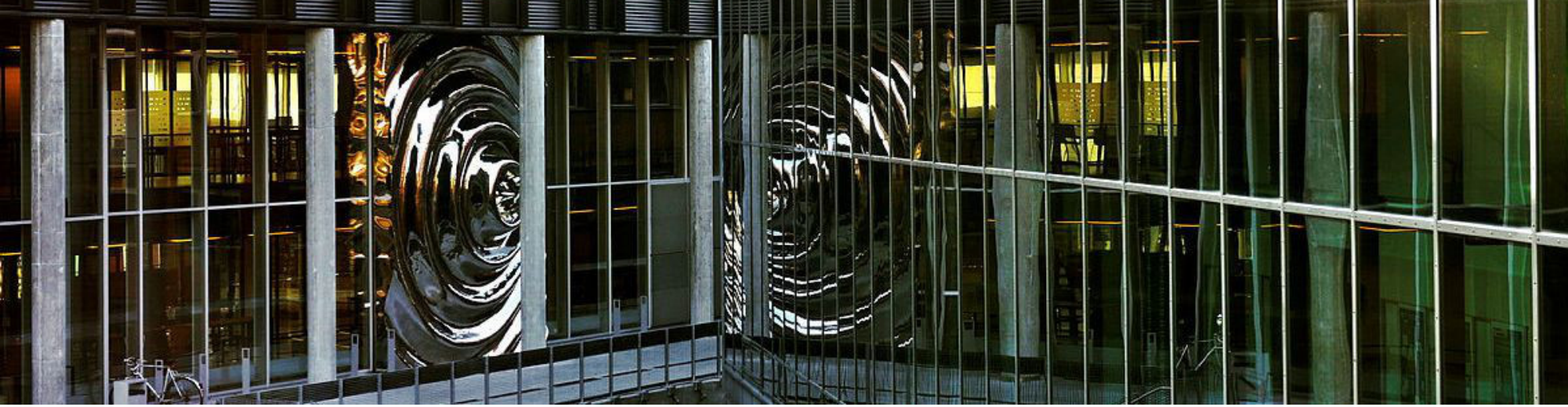

UiO : Department of Informatics
University of Oslo

Automatic Generation of Generators for Property-Based Testing with Inverse Interpretation

Joachim Tilsted Kristensen, Triera Gashi, and
Michael Kirkedal Thomsen,
michakt@ifi.uio.no

Reversible Computations July 10, 2026

Property-based testing

Property-based testing

Property-based testing

Property-based testing offers program validation by generating random inputs to validate behavioural program properties.

Most famous implementation is QuickCheck for Haskell

[Claessen and Hughes, 2000]

Property-based testing

Property-based testing

Property-based testing offers program validation by generating random inputs to validate behavioural program properties.

Most famous implementation is QuickCheck for Haskell

[Claessen and Hughes, 2000]

- Writing properties is hard...

Property-based testing

Property-based testing

Property-based testing offers program validation by generating random inputs to validate behavioural program properties.

Most famous implementation is QuickCheck for Haskell

[Claessen and Hughes, 2000]

- Writing properties is hard...
- However, manually writing test-input generators is the hidden problem

Writing test-input generators remains a high barrier in industry

[Lampropoulos et al., 2017a]

Aim of this work

- Make a proof-of-concept programming language, pun, that supports property-based testing natively.
  - Should be similar to vanilla languages
- pun generates inputs automatically
  - based on inverse interpretation of its typing judgement rules

Example 1

Consider, for instance, the following property:

```
property plus0-id f n . (f (n) + 0) == (f (n + 0)).
```

Example 1

Consider, for instance, the following property:

```
property plus0-id f n . (f (n) + 0) == (f (n + 0)).
```

This is a well-formed property since

- Right-hand side returns type `boolean`
- `f` has type `integer -> integer`
- `n` has type `integer`

Example 1

Consider, for instance, the following property:

```
property plus0-id f n . (f (n) + 0) == (f (n + 0)).
```

This is a well-formed property since

- Right-hand side returns type `boolean`
- `f` has type `integer -> integer`
- `n` has type `integer`

Result is

```
$ pun --check examples/rc1.pun
testing:plus0-id> ........................... ok
```

Example 2

Consider, for instance, the following property:

```
property plus1-id f n . (f (n) + 1) == (f (n + 1)).
```

Example 2

Consider, for instance, the following property:

```
property plus1-id f n . (f (n) + 1) == (f (n + 1)).
```

This is a well-formed property since

- Right-hand side returns type `boolean`
- `f` has type `integer -> integer`
- `n` has type `integer`

Example 2

Consider, for instance, the following property:

```
property plus1-id f n . (f (n) + 1) == (f (n + 1)).
```

This is a well-formed property since

- Right-hand side returns type `boolean`
- `f` has type `integer -> integer`
- `n` has type `integer`

Result is

```
pun --check examples/rc2.pun
testing:plus1-id> .x failed:
shrinking> (((\x -> x + x)(17)) + 1) ==
             ((\x -> x + x)(17 + 1))
after 1 tests.
```

Approach

This is not the first attempt to aid the programmer by creating the generators

[Lampropoulos et al., 2017b]

Foundation on generating well-typed terms

[Frank et al., 2024, Claessen et al., 2014].

Automatically generate test inputs following the random judgements approach presented. Generate well-typed expressions from standard judgement rules

[Fetscher et al., 2015]

Approach

This is not the first attempt to aid the programmer by creating the generators

[Lampropoulos et al., 2017b]

Foundation on generating well-typed terms

[Frank et al., 2024, Claessen et al., 2014].

Automatically generate test inputs following the random judgements approach presented. Generate well-typed expressions from standard judgement rules

[Fetscher et al., 2015]

Generate inputs automatically based on inverse interpretation of its typing judgement rules.

Generation based on inverse interpretation

Generate inputs automatically based on inverse interpretation of its typing judgement rules.

Generation based on inverse interpretation

Generate inputs automatically based on inverse interpretation of its typing judgement rules.

High quality testing should be likely to expose bugs in erroneous programs.

- For predicate `p`, generate a witness `i` such that the proposition `p(i) = False` holds.

Generation based on inverse interpretation

Generate inputs automatically based on inverse interpretation of its typing judgement rules.

High quality testing should be likely to expose bugs in erroneous programs.

- For predicate `p`, generate a witness `i` such that the proposition `p(i) = False` holds.

This suggests an inverse interpretation of testing, where faulty inputs are derived from a fixed undesirable output, namely `False`.

What is the problem with these generators?

Previous example in Haskell

```
plus0_id :: Fun Int Int -> Int -> Bool
plus0_id (Fun _ f) n = f n + 0 == f (n + 0)
```

Here the function is a data structure `Fun Int Int`.

What is the problem with these generators?

Previous example in Haskell

```
plus0_id :: Fun Int Int -> Int -> Bool
plus0_id (Fun _ f) n = f n + 0 == f (n + 0)
```

Here the function is a data structure `Fun Int Int`.

This is needed as we need an instance for the type class `Arbitrary`.

```
instance (Function a, CoArbitrary a, Arbitrary b)
   => Arbitrary (Fun a b)
```

What is the problem with these generators?

Previous example in Haskell

```
plus0_id :: Fun Int Int -> Int -> Bool
plus0_id (Fun _ f) n = f n + 0 == f (n + 0)
```

Here the function is a data structure `Fun Int Int`.

This is needed as we need an instance for the type class `Arbitrary`.

```
instance (Function a, CoArbitrary a, Arbitrary b)
   => Arbitrary (Fun a b)
```

- `b` needs to be an instance of `Arbitrary` because we need to be able to generate the outputs of our arbitrarily generated functions
- We must ensure that we generate equal outputs on equal inputs.

QuickCheck normally guarantees the latter by choosing the seed based on its input.

`Arbitrary` can be hard to implement I

Example working on binary search trees. [Hughes, 2019]
In Haskell

```haskell
data BST = Leaf | Node BST Int Int BST

instance Arbitrary BST where
    arbitrary =
      oneof [ return Leaf
            , insert <$> arbitrary <*>
                arbitrary <*> arbitrary
            ]

prop_InsertValid :: Int -> Int -> BST -> Bool
prop_InsertValid k v t = valid (insert k v t)

prop_FindPostPresent :: Int -> Int -> BST -> Property
prop_FindPostPresent k v t =
  find k (insert k v t) === Just v
```

`Arbitrary` can be hard to implement II

In pun

```
data BST = Leaf | Node [BST, integer, integer, BST] .

property insert-valid k v t .
  if   valid t
  then valid (insert k v t)
  else true .

property find-post-present k v t .
  if   valid t
  then (find k (insert k v t)) == (Just [v])
  else true .
```

Generates all possible tree and not just BST.

The pun Programming Language

The pun Programming Language Syntax I

The language pun is a higher-order functional language that uses call-by-value parameter passing.

$$
\begin{aligned}
\Delta ::= & \texttt{data}\ x = (\mathcal{C}[\tau_i])_j\ .\ \Delta \mid \texttt{property}\ x\ x_i\ .\ t\ .\ \Delta \\
& \mid f : \tau\ .\ \Delta \mid f x_i = t\ .\ \Delta \mid \epsilon \\
t ::= & \ x \mid \overline{n} \mid \textbf{true} \mid \textbf{false} \mid \mathcal{C}[t_i] \mid t_0 + t_1 \mid t_0 \leq t_1 \mid \lambda x.t_0 \mid t_1\ t_2 \\
& \mid \textbf{case}\ t\ \textbf{of}\ ;\ p_i \rightarrow t_i \mid \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \mid \textbf{rec}\ x.t \\
p ::= & \ x \mid \overline{n} \mid \textbf{true} \mid \textbf{false} \mid \mathcal{C}\ [p_i] \\
c ::= & \ \overline{n} \mid \textbf{true} \mid \textbf{false} \mid \mathcal{C}[c_i] \mid \lambda x.t_0 \\
\tau ::= & \ x \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \rightarrow \tau_2 \mid (\tau_1 \times \tau_2)
\end{aligned}
$$

- algebraic data type definition: keyword `data`
- property: keyword `property`
- Function definitions

pun Typing I

The typing system is fairly standard.

$\boxed{\Delta\Gamma \vdash t : \tau}$

$$\text{Lookup} : \frac{}{\Delta\Gamma \vdash x : \tau}\,(\Gamma(x) = \tau) \qquad \text{Number} : \frac{}{\Delta\Gamma \vdash \overline{n} : \texttt{int}}$$

$$\text{True} : \frac{}{\Delta\Gamma \vdash \textbf{true} : \texttt{bool}} \qquad \text{False} : \frac{}{\Delta\Gamma \vdash \textbf{false} : \texttt{bool}}$$

$$\tau\text{-Constructor} : \frac{\Delta\Gamma \vdash t_i : \tau_i}{\Delta[\textbf{data}\ \tau = (\mathcal{C}\ [\tau_i])_j]\Gamma \vdash \mathcal{C}\ [t_i]\ : \tau}$$

$$\text{LessThanOrEqual} : \frac{\Delta\Gamma \vdash t_0 : \texttt{int} \quad \Delta\Gamma \vdash t_1 : \texttt{int}}{\Delta\Gamma \vdash t_0 \leq t_1 : \texttt{bool}}$$

$$\text{Addition} : \frac{\Delta\Gamma \vdash t_0 : \texttt{int} \quad \Delta\Gamma \vdash t_1 : \texttt{int}}{\Delta\Gamma \vdash t_0 + t_1 : \texttt{int}}$$

$$\text{Abstraction} : \frac{\Delta\Gamma[x \mapsto \tau_1] \vdash t_0 : \tau_2}{\Delta\Gamma \vdash \lambda x.t_0 : \tau_1 \to \tau_2}$$

pun Typing II

$$\text{Application} : \frac{\Delta\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Delta\Gamma \vdash t_2 : \tau_1}{\Delta\Gamma \vdash t_1\ t_2 : \tau_2}$$

$$\text{Let} : \frac{\Delta\Gamma \vdash t_1 : \tau_1 \quad \Delta\Gamma[x \mapsto \tau_1] \vdash t_1 : \tau_2}{\Delta\Gamma \vdash \textbf{let}\ x = t_1\ \textbf{in}\ t_2 : \tau_2}$$

$$\text{Recursion} : \frac{\Delta\Gamma[x \mapsto \tau] \vdash t_0 : \tau}{\Delta\Gamma \vdash \textbf{rec}\ x.t_0 : \tau}$$

$$\text{Case} : \frac{\Delta\Gamma \vdash t_0 : \tau_1 \quad \Delta\Gamma[x_i \mapsto \tau_i] \vdash p_i : \tau_1 \quad \Delta\Gamma[x_i \mapsto \tau_i] \vdash t_i : \tau_2}{\Delta\Gamma \vdash \textbf{case}\ t_0\ \textbf{of}\ \ ;\ p_i \to t_i\ \ : \tau_2} \quad (x_i \in \text{free}(p))$$

Arbitrary Generator Generation

Arbitrary Generator Generation I

Implements an inverse-interpretation the pun typing rules.
Searches the typing rules to find possible constructs.
For example, consider

```
property plus0-id f n . (f (n) + 0) == (f (n + 0)).
```

Arbitrary Generator Generation I

Implements an inverse-interpretation the pun typing rules.
Searches the typing rules to find possible constructs.
For example, consider

```
property plus0-id f n . (f (n) + 0) == (f (n + 0)).
```

Here `f` must inhabit the type `int` $\rightarrow$ `int`

Arbitrary Generator Generation I

Implements an inverse-interpretation the pun typing rules.
Searches the typing rules to find possible constructs.
For example, consider

```
property plus0-id f n . (f (n) + 0) == (f (n + 0)).
```

Here `f` must inhabit the type $\texttt{int} \to \texttt{int}$

Thus it must be a construct of the typing rule

$$? : \frac{\begin{array}{c}?\\ \vdots\end{array}}{\Delta\Gamma \vdash \texttt{f} : \texttt{int} \to \texttt{int}}$$

Arbitrary Generator Generation I

Implements an inverse-interpretation the pun typing rules.
Searches the typing rules to find possible constructs.
For example, consider

```
property plus0-id f n . (f (n) + 0) == (f (n + 0)).
```

Here `f` must inhabit the type $\texttt{int} \to \texttt{int}$

Thus it must be a construct of the typing rule

$$? : \frac{\begin{array}{c}?\\ \vdots\end{array}}{\Delta\Gamma \vdash \texttt{f} : \texttt{int} \to \texttt{int}}$$

The only rule which matches this form is the Abstraction rule:

$$\text{Abstraction} : \frac{? : \dfrac{\begin{array}{c}?\\ \vdots\end{array}}{\Delta\Gamma[x \mapsto \texttt{int}] \vdash t_0 : \texttt{int}}}{\Delta\Gamma \vdash \texttt{f} : \texttt{int} \to \texttt{int}} \; (\texttt{f} = \lambda x.t_0)$$

Arbitrary Generator Generation I

From here, the search for a derivation has several options: Lookup, Number, Addition, Application, Let, Recursion and Case rules all could possibly conclude a derivation where the type is `int`.

For example, the Lookup rule applies because Γ contains the binding $[x \mapsto \texttt{int}]$

$$\text{Abstraction}: \frac{\text{Lookup}: \dfrac{}{\Delta\Gamma[x \mapsto \texttt{int}] \vdash x : \texttt{int}}(\Gamma[x \mapsto \texttt{int}](x) = \texttt{int})}{\Delta\Gamma \vdash \texttt{f} : \texttt{int} \to \texttt{int}}\ (\texttt{f} = \lambda x.x)$$

Arbitrary Generator Generation I

From here, the search for a derivation has several options: Lookup, Number, Addition, Application, Let, Recursion and Case rules all could possibly conclude a derivation where the type is `int`.

For example, the Lookup rule applies because Γ contains the binding $[x \mapsto \texttt{int}]$

$$\text{Abstraction}: \frac{\text{Lookup}: \dfrac{}{\Delta\Gamma[x \mapsto \texttt{int}] \vdash x : \texttt{int}}(\Gamma[x \mapsto \texttt{int}](x) = \texttt{int})}{\Delta\Gamma \vdash \texttt{f} : \texttt{int} \to \texttt{int}}\ (\texttt{f} = \lambda x.x)$$

Since the derivation is now closed, the generator can safely choose $\lambda x.x$ to instantiate `f`.

Arbitrary Generator Generation I

From here, the search for a derivation has several options: Lookup, Number, Addition, Application, Let, Recursion and Case rules all could possibly conclude a derivation where the type is `int`.

For example, the Lookup rule applies because Γ contains the binding $[x \mapsto \texttt{int}]$

$$\text{Abstraction}: \frac{\text{Lookup}: \dfrac{(\Gamma[x \mapsto \texttt{int}](x) = \texttt{int})}{\Delta\Gamma[x \mapsto \texttt{int}] \vdash x : \texttt{int}}}{\Delta\Gamma \vdash \texttt{f} : \texttt{int} \to \texttt{int}} (\texttt{f} = \lambda x.x)$$

Since the derivation is now closed, the generator can safely choose $\lambda x.x$ to instantiate `f`.

pun Generates for all possible construct of the form

```
Type -> Gen (Term Type)
```

Evaluation

Evaluation I

Bugs that are searched for in the benchmark.
Identical to the bugs in Hughes [Hughes, 2019]

Bug #	Description
#1	`insert` discards the existing tree, returning a single-node tree just containing the newly inserted value.
#2	`insert` fails to recognize and update an existing key, inserting a duplicate entry instead.
#3	`insert` fails to update an existing key, leaving the tree unchanged instead.
#4	`delete` fails to rebuild the tree above the key being deleted, returning only the remainder of the tree from that point on.

Evaluation II

Bugs that are searched for in the benchmark.
Identical to the bugs in Hughes [Hughes, 2019]

Bug #	Description
#5	Key comparisons reversed in `delete`; only works correctly at the root of the tree.
#6	`union` wrongly assumes that all the keys in the first argument precede those in the second.
#7	`union` wrongly assumes that if the key at the root of `t` is smaller than the key at the root of `t'`, then all the keys in `t` will be smaller than the key at the root of `t'`.
#8	`union` works correctly, except that when both trees contain the same key, the left argument does not always take priority.

Result I

Bugs found: ✓pun ✗[Hughes, 2019].

	Insert bugs			**Delete bugs**		**Union bugs**		
Validity properties	#1	#2	#3	#4	#5	#6	#7	#8
insert-valid		✓✗						
nil-valid								
delete-valid		✗						
union-valid		✓✗				✓✗	✓✗	
Post-conditions	#1	#2	#3	#4	#5	#6	#7	#8
insert-post	✓✗	✓✗	✓✗					
delete-post		✗		✓✗	✓✗		✓	
find-post-present		✓✗	✓✗					
find-post-absent		✗		✓	✓✗			
insert-delete-complete	✓	✓✗		✓✗				
union-post	✓	✓	✓			✓✗	✓✗	✓✗

Result II

	Insert bugs			Delete bugs		Union bugs		
Metamorphic prop	#1	#2	#3	#4	#5	#6	#7	#8
insert-insert-weak	✓✗							
insert-insert	✓✗	✓✗	✓✗					
insert-delete-weak				✓✗			✓	
insert-delete		✓✗	✓✗	✓✗			✓	
insert-union	✗	✗	✗			✓✗	✓✗	✓✗
delete-nil							✓	
delete-insert-weak			✓	✓✗			✓	
delete-insert	✓✗	✓✗	✓	✓✗	✓✗		✓	
delete-delete			✓	✓✗	✓✗			
delete-union	✓	✓✗	✓	✓✗	✓✗	✓✗	✓✗	✗
union-nil-1								
union-nil-2	✓						✓	
union-delete-insert	✓✗	✓✗	✓✗	✓✗	✓✗	✓✗	✓✗	✗
union-union-idem	✓	✓				✓✗		
union-union-assoc						✗	✓✗	✗
find-nil								
find-insert	✓✗	✓✗	✓✗					
find-delete	✓	✗		✓✗	✓✗		✓	
find-union	✓	✓	✓			✓✗	✓✗	✓✗

Result III

	Insert bugs			Delete bugs		Union bugs		
Preservation of equivalence	#1	#2	#3	#4	#5	#6	#7	#8
insert-preserves-equiv								
delete-preserves-equiv	✓✗			✓✗	✓✗	✓	✓	
union-preserves-equiv	✓✗					✓	✓✗	✗
find-preserves-equiv	✓✗							
Completeness of insertions	#1	#2	#3	#4	#5	#6	#7	#8
insert-complete	✓							
insert-complete-for-delete	✓							
insert-complete-for-union	✓✗					✓✗	✓✗	
Model-based properties	#1	#2	#3	#4	#5	#6	#7	#8
nil-model								
insert-model	✓✗	✓✗	✓✗					
delete-model	✓	✗		✓✗	✓✗		✓	
union-model		✗				✗	✗	✗
find-model								
Total failure	21/	15/	13/	13/	9/	10/	19/	3/
pun / [Hughes, 2019]	12	17	8	12	9	10	10	8

Conclusion

Conclusion

Contributions:

- Defined a small language pun that can automatically generates generators.
- Extended the work of [Fetscher et al., 2015] with inverse interpretation of the falsified properties.
- Show that the approach has comparable effectiveness to hand-made generators [Hughes, 2019].
  - At the cost of performing more tests.
  - 100,000 tests still only takes seconds (Haskell standard is 100)

Future Work:

- Improved search for counter examples.

Thank you

The implementation of pun, and the benchmarks can be found at https://github.com/jtkristensen/pun-lang.

Thank you

The implementation of pun, and the benchmarks can be found at https://github.com/jtkristensen/pun-lang.

Questions

?

Bibliography I

Claessen, K., Duregård, J., and Pałka, M. H. (2014).

Generating constrained random data with uniform distribution.
In Codish, M. and Sumii, E., editors, *Functional and Logic Programming*, pages 18–34. Springer International Publishing.

Claessen, K. and Hughes, J. (2000).

Quickcheck: A lightweight tool for random testing of Haskell programs.
In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279. Association for Computing Machinery.

Fetscher, B., Claessen, K., Pałka, M., Hughes, J., and Findler, R. B. (2015).

Making random judgments: Automatically generating well-typed terms from the definition of a type-system.
In *Programming Languages and Systems*, pages 383–405. Springer Berlin Heidelberg.

Frank, J., Quiring, B., and Lampropoulos, L. (2024).

Generating well-typed terms that are not "useless".
Proc. ACM Program. Lang., 8(POPL).

Hughes, J. (2019).

How to specify it! A guide to writing properties of pure functions.
In *Trends in Functional Programming: 20th International Symposium, TFP 2019*, pages 58–83. Springer-Verlag.

Lampropoulos, L., Gallois-Wong, D., Hriţcu, C., Hughes, J., Pierce, B. C., and Xia, L.-y. (2017a).

Beginner's luck: A language for property-based generators.
In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 114–129. Association for Computing Machinery.

Bibliography II

Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. (2017b).
Generating good generators for inductive relations.
Proceedings of the ACM on Programming Languages, 2(POPL):1–30.

UiO : Department of Informatics
University of Oslo

Joachim Tilsted Kristensen,
Triera Gashi, and
Michael Kirkedal Thomsen,

Automatic Generation of Generators for Property-Based Testing with Inverse Interpretation