QuickChecking Convergence of Rewriting Systems (Functional Pearl)

QuickChecking Convergence of Rewriting Systems (Functional Pearl)

KOEN CLAESSEN, Chalmers University of Technology and University of Gothenburg, Sweden

Term rewriting systems are a common tool in automated reasoning and semantics of programming languages, and many practical applications require these systems to be convergent. While automated tools and theory exist to establish convergence, this paper is concerned with a practical method for testing it to quickly find useful counterexamples. Standard property-based testing approaches struggle here: exhaustively computing all normal forms is fundamentally flawed and too slow, while generating random normal forms makes counterexample minimization (shrinking) fragile due to dependencies on earlier generated test data. To solve this, we introduce a QuickCheck testing method based on generating and shrinking random execution traces. By checking if the first and last terms of a generated trace share the same deterministic normal form, we remove the data dependency between generators. This approach yields a property that efficiently finds counterexamples and enables fast, robust shrinking. We demonstrate the effectiveness of this method on various examples, ranging from group theory equations to distributed process calculus.

CCS Concepts: • **Software and its engineering** → *Semantics*; *Software testing and debugging*.

Additional Key Words and Phrases: Testing, Property-based testing, QuickCheck, shrinking, term rewriting

ACM Reference Format:
Koen Claessen. 2026. QuickChecking Convergence of Rewriting Systems (Functional Pearl). *Proc. ACM Program. Lang.* 10, ICFP, Article 280 (August 2026), 14 pages. https://doi.org/10.1145/3828678

1 Introduction

Term rewriting systems (TRS) are a common tool in automated reasoning and semantics of programming languages. A TRS is specified by a recursively defined set of *terms* and a (typically non-deterministic) *step* relation that can be applied anywhere in the term. Rewriting a term consists of applying possibly multiple steps in sequence to a term *t1* resulting in a new term *t2*, denoted $t1 \longrightarrow\!\!* \; t2$.

Common questions that arise when performing rewriting are:

- *Termination* (also called *strong normalization*): Starting from a term *t1*, must the rewriting process always end with a term *t2* that cannot be rewritten? A term *t2* like that is called a *normal form*.
- *Confluence* (also called the *diamond property*): If a term *t* can be rewritten to two different terms *t1* and *t2*, can we find a third term *t3* such that $t1 \longrightarrow\!\!* \; t3$ and $t2 \longrightarrow\!\!* \; t3$?

If a TRS is terminating and confluent, we call it *convergent*. Convergence means that every term *t* has a unique normal form [2].

Many practical applications of rewriting systems require a system to be convergent. For example, in automated equational reasoning, the aim is to automatically prove $t1 = t2$ from a set T of equational axioms. One method to establish these proofs is to construct a convergent TRS such that $t1 = t2$ if and only if we can find a term *t3* such that $t1 \longrightarrow\!\!* \; t3$ and $t2 \longrightarrow\!\!* \; t3$. The process of building

Author's Contact Information: Koen Claessen, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden, koen@chalmers.se.

ACM 2475-1421/2026/8-ART280
https://doi.org/10.1145/3828678

this convergent TRS is called *completion* [9] and it's at the heart of modern high-performance equational theorem provers [12].

Another application of rewriting systems is in programming language semantics. The operational semantics of a programming language can be specified in terms of a term rewriting system. For example, Core Verse (the core language of the programming language Verse) has been given semantics like this [1]. Even though the rewriting system of a general programming language is not convergent (because it should allow for non-terminating programs), we can often define subsystems that are expected to be convergent (for example by disallowing computation steps that unfold recursion). Convergence is important for languages that are supposed to be deterministic.

It is also common for languages to have a so-called *small step operational semantics*, where a computation step is defined in terms of a configuration c_1 taking one step to another configuration c_2, written $c_1 \longrightarrow c_2$. Sometimes it is the case that the step relation is non-deterministic, but the overall evaluation process is supposed to be deterministic. An example is a distributed system, where processes communicate with each other by message passing. The internal order of events may be non-deterministic, but the final outcome may be desired to be deterministic. An example is discussed in Sect. 3. Although not strictly a *term* rewriting system, they can very much be treated as such by the methods described in this paper. These kinds of systems are therefore sometimes called *abstract rewriting systems* (ARS). An ARS has the same notion of step, normal form, termination, confluence, and converge as a TRS; the only difference is that an ARS does not necessarily work on a recursively defined term structure. We do not treat TRS and ARS differently in this paper.

There exists a huge body of theory that can help establish termination, confluence, and non-confluence of rewriting systems [2], and even automated tools [7]. This paper is concerned with a practical method for *testing* convergence of a given rewrite system. The focus is on a practical method that can quickly find useful counterexamples to convergence. Here are the requirements we have on such a method:

- (R0) There is no restriction on the formalism. The rewriting system is assumed to be simply specified by a function $step :: Term \to [\,Term\,]$. Here, *Term* can be any type and use any datastructures available to us from the host language, and *step* can use any computable function to implement matching, side conditions, and new right-hand sides on these datastructures. (This requirement makes it hard to *analyze* rewriting rules as first-class objects; for example we cannot compute and evaluate *critical pairs* [2].)
- (R1) Each test should run quickly, so we can run many tests. We also want tests to be effective at finding bugs, and sometimes that is in conflict with a single test running quickly!
- (R2) A failing test should produce a counterexample that is easy to understand. Counterexample minimization, also called *shrinking* in the QuickCheck [3] community, should be compatible with the test method. For convergence testing, this turns out to be surprisingly tricky.

A note on non-termination. This paper focuses on testing whether every term has a unique normal form. It does not aim to directly test termination. In general, we cannot decide if rewriting a term does not terminate. So, throughout the paper, we assume that either the systems we deal with are in fact terminating, or that it's detectable by a human when they are not (for example, rewriting a term may take an unreasonably long time).

2 Motivating Example: Group Theory

This section introduces the motivating example we use to illustrate the way we test convergence of rewriting systems. It is adapted from the classic paper on completion by Knuth and Bendix [9].

$$
\begin{aligned}
e * x &= x \\
(x * y) * z &= x * (y * z) \\
inv\ x * x &= e
\end{aligned}
$$

Fig. 1. Group theory equations

$$
\begin{aligned}
e * x &\longrightarrow x \\
x * e &\longrightarrow x \\
inv\ e &\longrightarrow e \\
inv\ (inv\ x) &\longrightarrow x \\
(x * y) * z &\longrightarrow x * (y * z) \\
inv\ x * x &\longrightarrow e \\
inv\ x * (x * y) &\longrightarrow y \\
x * (inv\ x * y) &\longrightarrow y \\
inv\ (x * y) &\longrightarrow inv\ y * inv\ x
\end{aligned}
$$

Fig. 2. Group theory rules (incomplete!)

Take a look at Fig. 1, where we show the three group theory axioms involving the group theory operations unit e, inverse inv, and multiplication $*$. Imagine we want to implement an algorithm that can prove, given two expressions *t1* and *t2* involving group theory operations, whether or not $t1 = t2$ using the group theory axioms.

One way to do this is to create a TRS with the following three conditions: (1) each rule in the TRS is implied by the group theory axioms, (2) the TRS establishes all three group theory axioms by normalizing both sides of each equation to the same normal form, and (3) the TRS is convergent. Knuth and Bendix describe an algorithm for computing such a TRS, but we've made a first try in Fig. 2. We can easily check conditions (1) and (2), but is the TRS convergent?

Let's find out by implementing the TRS in Haskell and *testing* convergence.

2.1 Implementing Group Theory

We start by defining a datatype for terms involving the group theory operations. We also add variables to be able to represent general terms.

```
data Term
  = Unit   -- e
  | Inv Term
  | Term :*: Term
  | Var String
deriving (Eq, Ord)

instance Show Term where
  showsPrec _ Unit       = showString "e"
  showsPrec _ (Var x)    = showString x
  showsPrec p (Inv x)    = showParen (p ≥ 5) (showString "inv " ∘ showsPrec 5 x)
  showsPrec p (x :*: y) = showParen (p ≥ 4) (showsPrec 4 x ∘ showString "*" ∘ showsPrec 4 y)
```

For completeness, and to understand the output produced in the later examples, we show the full details of the Show instance.

Then, we define a function $top :: Term \to [Term]$ that defines all rewriting steps we may take at the very top-level of a term. Note that we cannot simply use pattern matching or use a big case expression to define *top*, because in general several rules may match the same term and so we can get several possible results.

```
top :: Term → [Term]
top lhs =
    [x                 | Unit :*: x          ← [lhs]]
  ++[x                 | x :*: Unit          ← [lhs]]
  ++[Unit              | Inv Unit            ← [lhs]]
  ++[x                 | Inv (Inv x)         ← [lhs]]
  ++[x :*: (y :*: z)   | (x :*: y) :*: z     ← [lhs]]
  ++[Unit              | Inv x :*: x'         ← [lhs], x ≡ x']
  ++[y                 | Inv x :*: (x' :*: y) ← [lhs], x ≡ x']
  ++[y                 | x :*: (Inv x' :*: y) ← [lhs], x ≡ x']
  ++[Inv y :*: Inv x   | Inv (x :*: y)       ← [lhs]]
```

We opted for list comprehensions as a good compromise between making the code for the rewrite rules look as close to the definition of the rules in Fig. 2 as possible. A rule of the shape *t1* ⟶ *t2* **if** *cond* is written as:

```
[t2 | t1 ← [lhs], cond]
```

To implement the *step* function, we need to not only apply the *top* function at top-level, but everywhere in the term. We use a helper function, *rec*, that applies its argument function once to each of the immediate recursive children[1]. In this way, *step* gathers all possible ways of making one rewrite step in the term.

```
step :: Term → [Term]
step t = top t ++rec step t

rec :: (Term → [Term]) → Term → [Term]
rec f (Inv x)   = [Inv x'   | x' ← f x]
rec f (x :*: y) = [x' :*: y | x' ← f x] ++[x :*: y' | y' ← f y]
rec _ _         = []
```

This way of defining rewrite rules on recursive terms goes back to Paulsson [10] and Visser [14].

2.2 Testing Convergence

So how do we test convergence? We are going to use QuickCheck [4], a popular property-based random testing tool. The idea is to write a *property*, a function that, given a term *t*, checks if *t* has more than one normal form, and then use *random test data generation* to check the property for randomly generated terms.

In QuickCheck, random test data generation and counterexample minimization are governed by the *Arbitrary* typeclass, which looks like:

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a → [a]
  shrink _  = []
```

The *arbitrary* method defines a monadic random generator for creating terms. The *shrink* method takes a failing test case and structurally produces a list of smaller candidate values to isolate the minimal bug. (For now, we focus on generating terms and defer structural shrinking to Sect. 2.4.)

[1]The function *rec* is definable generically in many generic programming libraries. We decided to implement it here directly for clarity.

The test data generator should generate random terms of type *Term*. The one thing we need to be careful about is termination of test data generation for recursive types. QuickCheck provides an intended *size* for generators that we decrease when recursively generating subterms.

```
instance Arbitrary Term where
  arbitrary = sized arb
    where
      arb n = frequency
                [ (1, return Unit)
                , (1, Var `fmap` elements ["x", "y", "z"])
                , (n, Inv `fmap` arb n1)
                , (n, liftM2 (:*:) (arb n2) (arb n2))
                ]
        where
          n1 = n - 1
          n2 = n `div` 2
```

We use the frequency combinator that randomly chooses between the given alternatives. We let the relative likelihood of choosing a recursive constructor (n) directly depend on the intended test data size. We also decide on a fixed set of variables that can occur in random terms.

How about the property? We want to check how many normal forms a given term has, so let's define a helper function *norms* that computes all normal forms of a given term t:

```
norms :: Term → [Term]
norms t = go empty [t]
  where
    go seen [ ]                = [ ]
    go seen (t : ts)
        | t `member` seen = go seen ts
        | otherwise       = case step t of
                              [ ] → t : go seen' ts
                              ts' → go seen' (ts' ++ ts)
      where
        seen' = insert t seen
```

The local *go* function keeps track of terms already encountered in the set *seen*. It keeps exploring the children of a term t in a depth-first manner, until there are no children, in which case it has found a normal form.

Now we are ready to define our first convergence property:

```
prop_Convergence1 t =
  case norms t of
    _ : _ : _ → False
    _         → True
```

We use *norms* to (lazily) compute all normal forms of t, and if there are at least 2, the property is *False*.

Let's QuickCheck the property!

```
GHCi> quickCheck prop_Convergence1
(6 tests)^C
```

After 5 tests, running the 6th, the property seems stuck. After a while we interrupt.

2.3 Random Rewriting

What happened? It turns out that the function *norms* is extremely slow. Even for terms that only have one normal form, it explores all possible ways a term can be rewritten. If there are (seemingly) independent parts of a term that can be rewritten, *norms* will consider all possible combinations of those rewritings. And it should, because it does not know whether these rewrites are actually independent or not! So, using the function *norms* is fundamentally flawed and breaks (R1) from the introduction.

What now? Instead of computing all normal forms, let's take the QuickCheck approach, and just compute two random normal forms. If they are not the same, then we've found a counterexample to convergence!

First, we define a generator for random normal forms:

```
arbNorm :: Term → Gen Term
arbNorm t = case step t of
  [] → return t
  ts → do t' ← elements ts
          arbNorm t'
```

Given a term, it checks if a step can be made. If not, we have a normal form. Otherwise, we take a random step and recurse. If the TRS at hand terminates, so does this generation.

Let's write a new property:

```
prop_Convergence2 :: Term → Property
prop_Convergence2 t =
  forAll (arbNorm t) $ λt1 →
    forAll (arbNorm t) $ λt2 →
      t1 ≡ t2
```

For any term t, and any two normal forms *t1* and *t2* of t, *t1* and *t2* must be equal.

Let's test it!

```
GHCi> quickCheck prop_Convergence2
*** Failed! Falsified (after 64 tests):
(((inv (inv z) * (inv e * y)) * (inv (inv (y * z)) * e)) * (z * (inv (inv z) * e))) * inv ((((x * inv
        x) * e) * (inv (inv e) * inv e)) * inv (inv y))
z * (y * (y * (z * (z * (z * (inv y * (x * inv x)))))))
z * (y * (y * (z * (z * (z * inv y)))))
```

Great! We found a counterexample. We now know that our TRS is not convergent. But the counterexample is really hard to understand. Let's use counterexample minimization, or *shrinking*.

2.4 Shrinking

The thing we must do is to specify a function *shrink :: Term -> [Term]* that, given the term that failed a property, produces smaller versions of that term that are likely to also be counterexamples. Important to realize is that shrinking is a step-wise, greedy process, that happens in small steps (otherwise the search space would be too large). Typically, we shrink a term to its immediate

subterms, or we recursively shrink one subterm. Depending on the domain, we often add extra shrinking rules. In this case, it is a good idea to also add a rewriting *step* as a shrinking step. If t can normalize to two different terms, then t' is likely to also do that for $t \longrightarrow t'$. Shrinking like this will bring us closer to the "fork" in the road that leads to two different normal forms.

The function *shrink* is actually a method of the *Arbitrary* class:

```
instance Arbitrary Term where
  ...
  shrink t = children t ++ top t ++ rec shrink t

children :: Term → [Term]
children (Inv x)   = [x]
children (x :*: y) = [x, y]
children _         = []
```

When adding domain-specific shrinking rules it's important to make sure that the shrinking process still terminates. In this case, that is unproblematic. We use the function *top* and reuse the function *rec* from earlier. In fact, term rewriting and shrinking share the trait that we want to apply one (rewrite / shrinking) step to any of the subterms!

Let's test the property again and see what happens.

```
GHCi> quickCheck prop_Convergence2
*** Failed! Falsified (after 52 tests and 10 shrinks):
inv ((y * inv (e * x)) * (x * (inv (e * y) * inv (e * x))))
x * (y * inv y)
x
```

Much better! To understand this counterexample, t is printed on the first line, and *t1* and *t2* on the following two lines.

However, it's still hard to see what the problem actually is. And if we look at the term found for t, it still seems to have irrelevant subterms such as $e * x$ and $inv\ (e * y)$ that surely should be reduced to simpler terms?

The problem is this. In the property, *t1* and *t2* are *dependent* on the value chosen for t. If we change the value of t to another term during shrinking, the property will generate two completely new random normal forms of t. There is a certain probability that these two random normal forms are different, in which case the shrinking step succeeded, but it is more likely that those two randomly chosen normal forms are identical, especially if there is a very subtle case of non-convergence.

So, shrinking test data when several generators are involved that depend on earlier generated test data is hard and we cannot expect shrinking to reliably succeed when dependent generators are involved.

One solution to this problem that was used in earlier work on testing of race conditions using QuickCheck[5], is to, when shrinking t, to not only test the rest of the property once, but several times (say, 50 times), in order to be more robust, to make it more likely that if the new t has more than one normal form, we actually find it.

Here is one way to do that:

```
prop_Convergence3 :: Shrinking → Term → Property
prop_Convergence3 shr t =
  conjoin [ forAll (arbNorm t) $ λt1 →
              forAll (arbNorm t) $ λt2 →
```

```
        t1 ≡ t2
  | i ← [1 .. if shr ≡ Yes then 50 else 1]
  ]
```

This property uses the combinator *conjoin* that takes a list of properties and produces a property that tests all of them. We use a trick to detect if we are shrinking or not by adding an extra argument to the property, in which case we test the inner property 50 times instead of once.

QuickCheck this property gives:

```
GHCi> quickCheck prop_Convergence3
*** Failed! Falsified (after 36 tests and 7 shrinks):
z * (inv z * e)
e
z * inv z
```

A minimal counterexample! We can now see what is going on. We forgot a rule:

$$x * inv\ x \longrightarrow e$$

Even though *prop_Convergence3* found a good counterexample, we reject this solution. Firstly, because the number 50 is arbitrary, and using this solution, we always end up fiddling with this number. Secondly, in larger examples (such as the Core Verse language or testing race conditions), we found that shrinking becomes very expensive and is still not robust enough for hard-to-find counterexamples. We want to understand how to shrink these kinds of properties principally.

2.5 Traces

One observation we can make about *prop_Convergence2* is that it is unnecessary to generate two random normal forms; we can also generate one deterministic normal form and one random normal form.

If we define:

```
norm :: Term → Term
norm = head ∘ norms
```

We always get the left-most normal form. (Note that *norms* is lazy and will produce its first argument quickly; computing if there are any more terms in the result is what costs.)

Then we can redefine *prop_Convergence2* as follows:

```
prop_Convergence2' :: Term → Property
prop_Convergence2' t =
  let t1 = norm t in
    forAll (arbNorm t) $ λt2 →
      t1 ≡ t2
```

The likelihood of this property failing is in practice the same as the likelihood of the original property failing (we found this by measuring on large TRS examples), but theoretically, in the worst case, we may need to run at most twice as many tests to get the same likelihood as the original property.

What do we gain then? We want to involve the *trace* going from t to $t2$ in the shrinking. We want shrinking to start by shrinking the length of that trace down to 1. If the length of the trace is 1, then we can shrink the term normally and use *step*.

Here is what we do concretely. We start by defining a generator for random traces. This is adapted from the definition of *arbNorm*.

```
arbTrace :: Term → Gen [Term]
arbTrace t = case step t of
  [] → return [t]
  ts → do t' ← elements ts
          (t:) `fmap` arbTrace t'
```

Then, we define a new type that helps us define shrinking over traces.

```
data Trace = Trace [Term] deriving (Eq, Ord, Show)

instance Arbitrary Trace where
  arbitrary = do t ← arbitrary
                 ts ← arbTrace t
                 return (Trace ts)
```

We don't define a shrinking function just yet.

Here is our first try at a property using traces.

```
prop_Convergence4' :: Trace → Bool
prop_Convergence4' (Trace ts) =
  norm (head ts) ≡ last ts        -- not quite right yet!
```

This property tests the same thing as *prop_Convergence2'* from earlier. However, there is only one argument generated now, a trace. There is no dependency of a later generator on an earlier one.

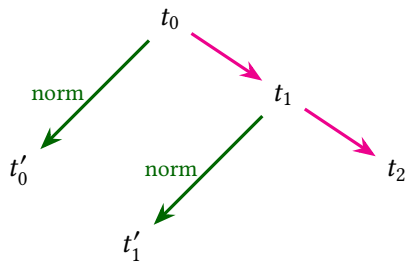

Fig. 3. Bisecting a counterexample trace $[t0, \ldots, t1, \ldots, t2]$ into $[t0, \ldots, t1]$ or $[t1, \ldots, t2]$.

How do we shrink a trace? Take a look at Fig. 3. Imagine if a found counterexample trace contains the terms: $[t0, \ldots, t1, \ldots t2]$. (Here, $t2$ is a normal form.) Since this is a counterexample, we know that $norm\ t0 \not\equiv t2$. Now consider $norm\ t1$. It cannot be equal to both $norm\ t0$ and $t2$, so, it must be different from either $norm\ t0$ or $t2$ (or both). Depending on which it is, we shrink to the trace $[t0, \ldots, t1]$ or the trace $[t1, \ldots, t2]$.

With one detail missing; the trace $[t0, \ldots, t1]$ does not necessarily end in a normal form. So, we generalize our property to deal with all traces, even the ones that do not end in a normal form:

```
prop_Convergence4 :: Trace → Bool
prop_Convergence4 (Trace ts) =
  norm (head ts) ≡ norm (last ts)        -- final correct version
```

Checking if a trace is OK amounts to checking that the first and last term have the same (deterministic) normal form. If we only generate traces that end in a normal form, that second *norm* has no effect, but it will have an effect when we shrink.

The full shrinking function:

```
instance Arbitrary Trace where
  ...
  shrink (Trace [_])     = []
  shrink (Trace [t1, _]) = [Trace [t1', t2'] | t1' ← shrink t1, t2' ← step t1']
  shrink (Trace ts)      = [Trace (take (k + 1) ts), Trace (drop k ts)]
    where k = length ts `div` 2
```

The last line in effect uses binary search to shrink the trace. Once a trace is shrunk down to only one step, $[t1, t2]$, we shrink the actual term *t1* and consider all steps we can take from the result, ignoring the original *t2*.

So, shrinking a term t with two different normal forms first shrinks the trace to a one-step trace $t1 \longrightarrow t2$ such that *norm t1* and *norm t2* are different, and then applies regular shrinking to *t1* only.

QuickChecking this property gives:

```
GHCi> quickCheck prop_Convergence4
*** Failed! Falsified (after 67 tests and 11 shrinks):
x * (inv x * e)
e
x * inv x
```

Success! Efficient counterexample finding combined with fast and robust shrinking.

2.6 A Simpler Approach?

So, if we can always shrink to a term *t1* such that we have $t1 \longrightarrow t2$ and *norm t1* $\not\equiv$ *norm t2*, why don't we define a property like that from the beginning?

```
prop_Convergence5 :: Term → Bool
prop_Convergence5 t =
  all (λt' → norm t ≡ norm t') (step t)
```

A term t is OK if every term t' we can reach in one step has the same (deterministic) normal form as t.

The problem is that we are a lot less likely to find a counterexample using this approach! To find a counterexample, we are required to generate a random term that exhibits the problem already for the first step. Typically the problematic step (that switches normal form) happens later in a trace.

In our larger examples for term rewriting systems, we typically needed up to 100 times more tests to find counterexamples using *prop_Convergence5* (see Sect. 4).

The problem with *prop_Convergence5* becomes apparent when we test convergence of abstract rewriting systems (for example representing a distributed system, see next section), where we always generate t that are in the initial state of a system; *prop_Convergence4* will generate random traces of the system until termination and thus explore interesting scenarios. *prop_Convergence5* will only consider single steps that happen at top-level.

3 Abstract Rewriting Systems: Process Calculus

Property *prop_Convergence4* is not restricted to traditional term rewriting systems; it is equally applicable to general abstract rewriting systems (ARS) where reductions happen over state configurations rather than recursive terms [2]. Here, we use the property in testing distributed algorithms where independent processes communicate via synchronous message passing.

Consider a system where a set of processes are concurrently executing a consensus or routing protocol. While the message scheduling interleavings are highly non-deterministic, the macro-level outcome of the protocol is expected to be entirely deterministic (convergent) once the network quiesces.

We model this synchronous process calculus in Haskell as follows. A system state is captured as a *Map Pid Proc*, mapping process IDs to their current execution state:

```
type Pid = Int
type Msg = String

data Proc
  = Send Pid Msg Proc
  | Recv [(Msg, Proc)]
  | Stop
  deriving (Eq, Ord, Show)

step :: Map Pid Proc → [Map Pid Proc]
step procs =
  [delete pid procs                          | (pid, Stop) ← ps] ++
  [insert pid1 p1 (insert pid2 p2 procs) | (pid1,  Send pid2 msg p1) ← ps
                                         , (pid2', Recv msgs)        ← ps
                                         , pid2 ≡ pid2'
                                         , (msg', p2) ← msgs
                                         , msg ≡ msg' ]
where
  ps = toList procs
```

The step function produces all possible pairings of senders with their possible receivers.

To test convergence of a concrete protocol, the test data generator generates random initial configurations of the system (collections of processes executing the protocol), and passes these to the *prop_Convergence* properties.

Convergence testing was one of the main focuses of [5], where repeated testing while shrinking (the technique behind *prop_Convergence3*) was advocated as the solution. In our experience, *prop_Convergence4* has a much more robust shrinking behavior (see Sect. 4).

4 Experimental Results

To give an indication of the practical value of the various properties, we provide some empirical results on a collection of examples. These are all examples of rewrite systems that were supposed to be convergent, but due to a mistake (deliberate or real) are not. All examples are terminating rewrite systems.

- **Group theory** The running example in this paper, with one rule missing. Language size: 4 constructors. Rule system size: 9 rules.

- **Esterel** A rewrite system that implements the semantics of a subset of Esterel, akin to the rules in [8], with a simplification that was wrong. Language size: 10 constructors. Rule system size: 21 rules.
- **Autosubst** A Haskell port of Autosubst made by a student [11, 15]. The language used consists of several mutually recursive datatypes, and the bug was that one of the rules was not properly applied to all relevant occurrences in all datatypes. Language size: 22 constructors. Rule system size: 52 rules.
- **Verse** A Haskell implementation of a version of the Verse Core Calculus [1]. The bug was a missing side condition on a rule. Language size: 14 constructors. Rule system size: 32 rules (with very complex side conditions and matching on contexts).

Let's now see how each of the properties we discussed in this paper did on these examples.

We divide up the comparisons into *bug finding capability* and *shrinking capability*. A good property needs to be good at both.

Bug finding. Here, we found that *prop_Convergence2*, 3, and 4 were all equally good at finding counterexamples (in fact, *prop_Convergence2* and 3 are identical in that regard), whereas *prop_Convergence5* requires orders of magnitudes more tests to find a counterexample. In the following table, we report the median number of tests before a counterexample is found, over 20 separate runs of the property. (Running one test takes about the same time in each of these properties.)

Table 1. Median number of tests required to find a counterexample over 20 runs

	Group	**Esterel**	**Autosubst**	**Verse**
prop_Convergence1	–	–	–	–
prop_Convergence2, 3	54	83	240	505
prop_Convergence4	61	98	339	618
prop_Convergence5	70	334	–	–

The "–" in the table indicates that no counterexample was found within the allotted number of tests (10,000) for a significant number of property runs.

Shrinking. In the evaluation of shrinking, we exclude *prop_Convergence1* and 5 because of their failure to find counterexamples.

Table 2. Median counterexample size after shrinking (and number of shrinking steps to find it)

	Group	**Esterel**	**Autosubst**	**Verse**
prop_Convergence2	20 (27)	19 (34)	32 (65)	45 (99)
prop_Convergence3	6 (80)	9 (811)	15 (1677)	–
prop_Convergence4	6 (34)	9 (50)	7 (47)	10 (180)

(The size of a reported counterexamples is the size of the AST.) As we can see, *prop_Convergence2* does not perform much shrinking at all, *prop_Convergence3* manages smaller counterexamples but at the cost of many failed shrinking steps.

Conclusion. The property *prop_Convergence4* is good at finding counterexamples and minimizing them. No other evaluated property shares both of these traits.

5 Discussion and Conclusions

We have presented a general method for random testing of convergence of a rewriting system. The final property is efficient, and produces minimal counterexamples in an efficient and robust way. As reported in Sec. 4, we have evaluated our method on several different kinds of rewriting systems, some math inspired, some in the area of programming languages semantics, and some in distributed algorithms. In all cases, *prop_Convergence4* provided the quickest route to counterexamples combined with robust shrinking.

We can't stress the value enough of having an easy and quick to use method for finding easily understandable counterexamples to properties you believe to be true, not only when developing a complicated software system, but also when developing complicated theoretical definitions, such as the semantics of a programming language. Perhaps ideally, one would have developed these rewrite systems in a theorem prover such as Lean [6] or Rocq [13], but we found that the flexibility of being in a general programming language with a mature version of QuickCheck made developing these properties very quick and easy.

Perhaps the most exciting implication of this method is that testing and proving are not mutually exclusive. When implementing an equational theorem prover, one typically needs to be able to detect non-confluence of rules, and then add new rules to eliminate that non-confluence (completion). The standard way to perform completion is to enumerate all *critical pairs*, combination of two rules that look they could lead to non-confluence, and then process each of these critical pairs. For non-trivial problems, there may be millions of critical pairs, and only a fraction actually leads to non-confluence. Using *prop_Convergence4* would be a different way of detecting non-confluence in a theorem prover. We have a prototype implementation in Twee [12] that uses random generation of traces and shrinking to generate rules that eliminate non-confluence, and it can already prove some properties that regular Twee can't!

Ultimately, we think that the pattern of generating execution traces and bisecting them during shrinking is a highly reusable abstraction. It provides a robust tool to tame the complexity of any nondeterministic system that demands a deterministic outcome.

6 Future Work

Some rewrite systems have so-called *structural rules* that formally cause non-termination, but that we are allowed to freely apply anywhere, specifically before matching a rule and before checking confluence. Examples are commutativity, associativity, but also more programming language specific things such as moving scoping of a variable up and down a term. What structural rules have in common is that they are detectably not terminating, there are only a finite number of different terms they can produce (this is also called looping). Typically, one has to change the matching function to deal with these rules, which hampers flexibility and simplicity, which in turn makes it easy to introduce bugs. We are looking at a more general way of dealing with these structural rules. Just treat them as normal rules, and instead change the normalize function $norm :: Term \to Term$ to compute a strongly connected component in the rewriting graph and return that instead.

References

[1] Lennart Augustsson, Koen Claessen, Simon Peyton Jones, Tim Sweeney, et al. 2023. The Verse Calculus: a Core Calculus for Functional Logic Programming. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/3607845

[2] Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press. https://doi.org/10.1017/CBO9781139172752

[3] Koen Claessen. 2012. Shrinking and showing functions: (Functional Pearl). In *ACM SIGPLAN Symposium on Haskell.* https://doi.org/10.1145/2364506.2364516

[4] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/351240.351266
[5] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. 2009. Finding Race Conditions in Erlang with QuickCheck and PULSE. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/1596550.1596574
[6] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*. Springer International Publishing, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37
[7] Jürgen Giesl et al. 2014. Proving Termination of Programs Automatically with AProVE. In *International Joint Conference on Automated Reasoning (IJCAR)*. https://doi.org/10.1007/978-3-319-08587-6_13
[8] Spencer P. Florence, Shu-Hung You, Jesse A. Tov, and Robert Bruce Findler. 2019. A Calculus for Esterel: If Can, Can. If No Can, No Can. *Proceedings of the ACM on Programming Languages* 3, POPL (January 2019), 61:1–61:29. https://doi.org/10.1145/3290374
[9] Donald E. Knuth and Peter B. Bendix. 1970. Simple Word Problems in Universal Algebras. In *Computational Problems in Abstract Algebra*. https://doi.org/10.1007/978-3-642-81955-1_23
[10] Lawrence C. Paulson. 1987. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press. https://doi.org/10.1017/CBO9780511526602
[11] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015 (Lecture Notes in Computer Science, Vol. 9236)*. Springer, 359–374. https://doi.org/10.1007/978-3-319-22102-1_24
[12] Nick Smallbone. 2020. Twee: An Equational Theorem Prover. In *International Joint Conference on Automated Reasoning (IJCAR)*. https://doi.org/10.1007/978-3-030-79876-5_35
[13] The Rocq Development Team. 2026. *The Rocq Prover Reference Manual*. INRIA. https://doi.org/10.5281/zenodo.1003420 Formerly known as the Coq Proof Assistant.
[14] Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *International Conference on Rewriting Techniques and Applications (RTA) (Lecture Notes in Computer Science, Vol. 2051)*. Springer-Verlag, 357–362. https://doi.org/10.1007/3-540-45127-7_27
[15] Marius Weidner. 2025. Haskell code for modeling Autosubst. https://github.com/koengit/autosubst-test. GitHub repository.

Received 2026-02-20; accepted 2026-05-13