QuickCheck Dynamic

Testing liveness (and safety) properties of stateful systems

Maximilian Algehed
joint work with John Hughes and Ulf Norell (Quviq)
and Arnaud Bailly (IOG)

Property based testing

Test.QuickCheck

The QuickCheck manual gives detailed information about using QuickCheck effectively. You can also try https://begriffs.com/p
written by a user of QuickCheck.

To start using QuickCheck, write down your property as a function returning `Bool`. For example, to check that reversing a list t

```
import Test.QuickCheck

prop_reverse :: [Int] -> Bool
prop_reverse xs = reverse (reverse xs) == xs
```

You can then use QuickCheck to test `prop_reverse` on 100 random lists:

```
>>> quickCheck prop_reverse
+++ OK, passed 100 tests.
```

To run more tests you can use the `withMaxSuccess` combinator:

```
>>> quickCheck (withMaxSuccess 10000 prop_reverse)
+++ OK, passed 10000 tests.
```

To use QuickCheck on your own data types you will need to write `Arbitrary` instances for those types. See the QuickCheck r

Property based testing

- Generate random inputs
- Run program
- Check result
  - ✅ Generate more tests
  - ❌ Shrink failing inputs

```
prop_revAppend :: [Int] -> [Int] -> Bool
prop_revAppend xs ys =
  reverse (xs ++ ys) == reverse xs ++ reverse ys

ghci> quickCheck prop_revAppend
*** Failed! Falsified (after 3 tests and 2 shrinks):
[0]
[1]
```

Property based testing

```haskell
prop_roundtrip :: Message -> Bool
prop_roundtrip msg = decode (encode msg) == msg
```

```haskell
prop_inOrder :: Tree -> Bool
prop_inOrder t = isSorted (inOrderTraversal t)
```

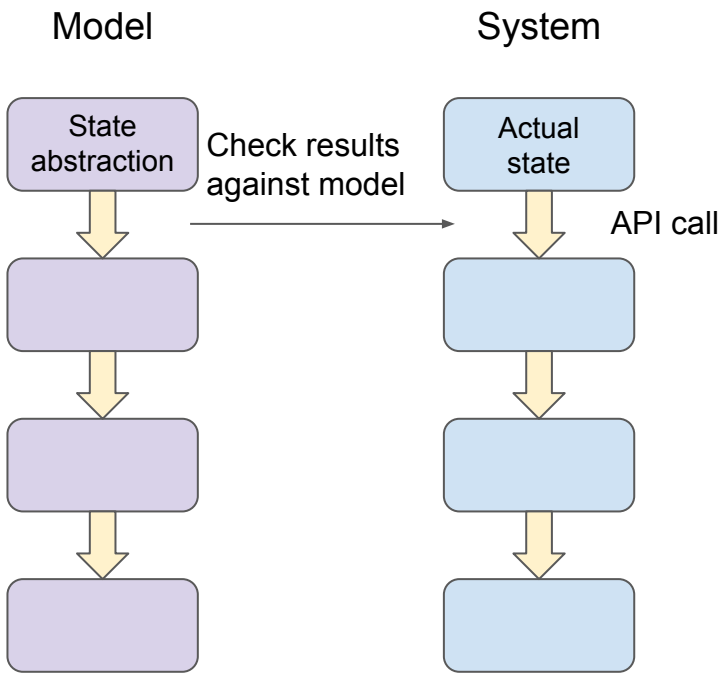
```haskell
prop_compileRun :: Program -> Bool
prop_compileRun p = run (compile p) == interpret p
```

Testing *stateful* systems

- What is the input?
  - ➢ Sequences of API calls
- How to check the output?
  - ➢ Compare to a *model* of the system

Testing stateful systems: model based testing

- **Abstract** actual state in a model state
- Model the **effect** of API calls on the model state
- Use model state to generate **sensible** calls
- **Compare** the actual execution with the model in a postcondition



Extra extra! Get it on Hackage today!

quickcheck-dynamic: A library for stateful property-based testing

[apache, library, testing] [Propose Tags]

Please see the README on GitHub at https://github.com/input-output-hk/quickcheck-dynamic#readme

[Skip to Readme]

Build InstallOk Documentation Available

Modules

[Index] [Quick Jump]

Test

QuickCheck

Test.QuickCheck.DynamicLogic
Test.QuickCheck.DynamicLogic.CanGenerate
Test.QuickCheck.DynamicLogic.Core
Test.QuickCheck.DynamicLogic.Quantify
Test.QuickCheck.DynamicLogic.SmartShrinking
Test.QuickCheck.DynamicLogic.Utils
Test.QuickCheck.StateModel

Downloads

- quickcheck-dynamic-2.0.0.tar.gz [browse] (Cabal source package)
- Package description (as included in the package)

Maintainer's Corner

For package maintainers and hackage trustees

- edit package information

Versions [RSS]
1.0.0, 1.1.0, **2.0.0**

Change log
CHANGELOG.md

Dependencies
base (>=4.7 && <5), mtl, QuickCheck, random [details]

License
Apache-2.0[multiple license files]

Author
Ulf Norell

Maintainer
arnaud.bailly@iohk.io

Category
Testing

Home page
https://github.com/input-output-hk/quickcheck-dynamic#readme

Bug tracker
https://github.com/input-output-hk/quickcheck-dynamic/issues

Source repo
head: git clone https://github.com/input-output-hk/quickcheck-dynamic

Uploaded
by abaillyiohk at 2022-10-11T09:06:24Z

Distributions
NixOS:1.1.0

Downloads

Test.QuickCheck.StateModel

```haskell
class StateModel state where

  data Action state a

  initialState    :: state
  precondition    :: state -> Action state a -> Bool
  nextState       :: state -> Action state a -> Var a -> state

  arbitraryAction :: VarContext -> state -> Gen (Any (Action state))
  shrinkAction    :: VarContext -> state -> Action state a -> [Any (Action state)]
```

Test.QuickCheck.StateModel

```haskell
type Env = forall a. Var a -> a

class Monad m => RunModel state m where
  perform       :: state -> Action state a -> Env -> m a
  postcondition :: (state, state) -> Action state a -> Env -> a -> Bool
```

- A model can have multiple backends
  - For instance, IO and IOSim

A simple example

```
whereis       :: MonadRegistry m => String -> m (Maybe ThreadId)
register      :: MonadRegistry m => String -> ThreadId -> m ()
unregister    :: MonadRegistry m => String -> m ()
```

- A Haskell toy implementation of the Erlang process registry
  - Threads can be assigned names (`register`, `unregister`)
  - and looked up by name (`whereis`)

A state model for the registry

```
type RegState = Map String (Var ThreadId)

instance StateModel RegState where
 data Action RegState a where
   Spawn      :: Action RegState ThreadId
   WhereIs    :: String -> Action RegState (Maybe ThreadId)
   Register   :: String -> Var ThreadId -> Action RegState ()
   Unregister :: String -> Action RegState ()


 nextState s (Register name tid) _ = s <> Map.singleton name tid
 nextState s (Unregister name) _   = Map.delete name s
 nextState s _ _                   = s
```

Note! Differs from Erlang registry!

A state model for the registry

```
instance RunModel RegState RegMonad where
  perform _ (Register name vTId) env = register reg name (env vTId)
  ...

  postcondition (s, _) (WhereIs name) env mTId =
    case Map.lookup name s of
      Nothing   -> mTId == Nothing
      Just vTId -> mTId == Just (env vTId)
  postcondition _ _ _ _ = True
```

A state model for the registry

```
ghci> quickCheck $ prop_stateModel @RegState
*** Failed! Falsified (after 5 tests and 1 shrink):
do var1 <- action $ Spawn
   action $ Register "a" var1
   action $ Register "b" var1
   -- Postcondition failed for the following action
   action $ Whereis "b"
```

A state model for the registry

```
type RegState = Map String (Var ThreadId)

instance StateModel RegState where
  data Action RegState a where
    Spawn      :: Action RegState ThreadId
    WhereIs    :: String -> Action RegState (Maybe ThreadId)
    Register   :: String -> Var ThreadId -> Action RegState ()
    Unregister :: String -> Action RegState ()

  nextState s (Register name tid) _
    | tid `notElem` Map.elements s = s <> Map.singleton name tid
  nextState s (Unregister name) _  = Map.delete name s
  nextState s _ _                  = s
```

A state model for the registry

```
ghci> quickCheck $ prop_stateModel @RegState
+++ Ok, passed 100 tests.
```

Liveness properties

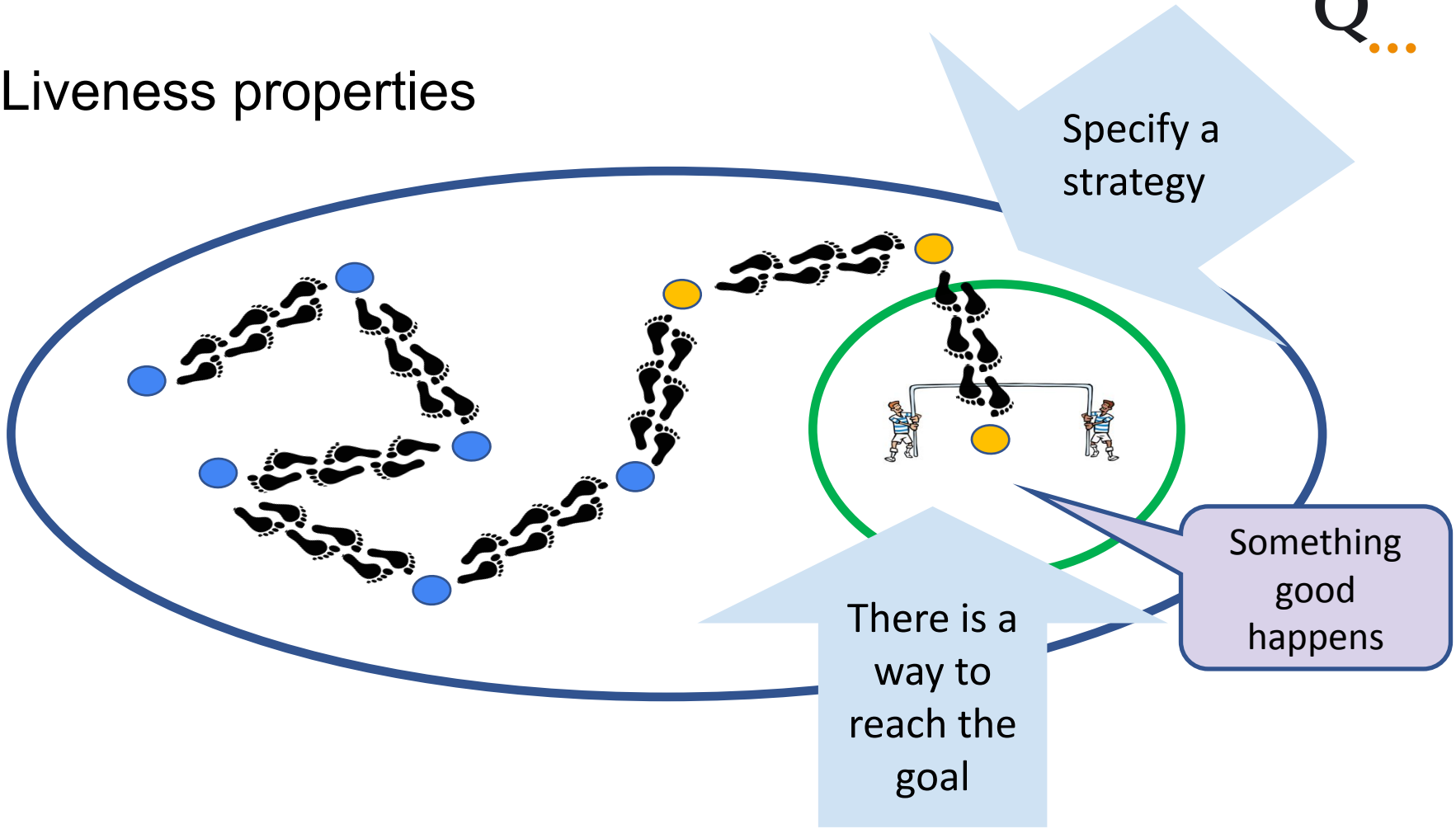
Whatever happens, the system is always alive
- something good eventually happens!

“The server **always eventually** responds to all requests”

“**There is always** some **sequence** of API calls that will unregister a given user”

“It’s **always possible** for participants in a smart contract to get their money out, **eventually**”

Liveness properties



Dynamic logic

```
anyActions_      :: DL state ()
```

Random walk

```
action           :: Action state a -> DL state (Var a)
```

Strategy

```
getModelStateDL :: DL state state
assert           :: String -> Bool -> DL state ()
```

Goal

```
forAllDL         :: StateModel => DL state () -> Property
```

Something good always happens eventually

Registry example

- Liveness property: can always register any thread under any name without breaking the system

```
prop_register :: (Var ThreadId -> String -> Script RegState a)
              -> Property
prop_register strat = forAllScript $ do
  anyActions_
  vTId <- pickRandomVar
  nm   <- pickRandomString
  strat vTId nm
  m <- getModelState
  assert "..." $ lookup vTId m == nm
  anyActions_
```

Registry example

```
strat0 vTId nm = action $ Register nm vTId
```

```
ghci> quickCheck $ prop_register strat0
*** Failed! Falsified (after 5 tests and 1 shrink):
do var2 <- action $ Spawn
   var3 <- action $ Spawn
   action $ Register "a" var3
   _ <- forAllQ $ exactlyQ $ "a"
   _ <- forAllQ $ exactlyQ $ var2
   action $ Register "a" var2
   assert "..." False
```

Registry example

```
strat1 vTId nm = do
  action $ Unregister nm
  action $ Register nm vTId
```

```
ghci> quickCheck $ prop_register strat1
*** Failed! Falsified (after 9 tests and 4 shrinks):
do var1 <- action $ Spawn
   action $ Register "a" var1
   _ <- forAllQ $ exactlyQ $ "d"
   _ <- forAllQ $ exactlyQ $ var1
   action $ Unregister "d"
   action $ Register "d" var1
   assert "..." False
```

Registry example

```haskell
strat2 vTId nm = do
  m <- getModelState
  case Map.findKeyOf vTId m of
    Just nm' -> action $ Unregister nm'
    Nothing  -> pure ()
  action $ Unregister nm
  action $ Register nm vTId
```

```
ghci> quickCheck $ prop_register strat2
+++ Ok, passed 100 tests.
```

What did we learn?

- Building a QuickCheck-dynamic model is straightforward
- QuickCheck showed us the way to a correctness proof
- Our property and strategy document themselves and our code

Applications

- Cryptocurrencies - IOG (Cardano Blockchain)
- Telecoms - Ericsson (telephone switches)
- Automotive - Volvo Cars
- Web - Fastly (edge computing / CDN)
- Health-care databases - NHS
- And many more! Come talk to us!