CODE BEAM EUROPE

STOCKHOLM

HYBRID CONFERENCE

Improve your tests with Makina

Luis Eduardo Bueso de Barrio

May 20 | 2022

#CodeBEAM

Before:

files	blank	comment	code
4	760	383	4513

After:

files	blank	comment	code
18	500	408	1692

PBT models

PBT models

Proprety-Based Testing (PBT) is a great testing methodology.

Successful tools widely used:

- Quviq QuickCheck
- PropEr

PBT models

Proprety-Based Testing (PBT) is a great testing methodology.

Successful tools widely used:

- Quviq QuickCheck
- PropEr

These tools are great for testing pure functions.

PBT models

Proprety-Based Testing (PBT) is a great testing methodology.

Successful tools widely used:

- Quviq QuickCheck
- PropEr

These tools are great for testing pure functions.

They have mechanisms to test stateful programs.

PBT state-machines or models.

A PBT model works like an oracle.

PBT models

Proprety-Based Testing (PBT) is a great testing methodology.

Implementation Model

Successful tools widely used:

- Quviq QuickCheck
- PropEr

These tools are great for testing pure functions.

They have mechanisms to test stateful programs.

PBT state-machines or models.

A PBT model works like an oracle.

PBT models

Proprety-Based Testing (PBT) is a great testing methodology.

Successful tools widely used:

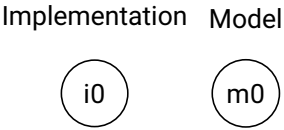
- Quviq QuickCheck
- PropEr

These tools are great for testing pure functions.

They have mechanisms to test stateful programs.

PBT state-machines or models.

A PBT model works like an oracle.



PBT models

Proprety-Based Testing (PBT) is a great testing methodology.

Successful tools widely used:

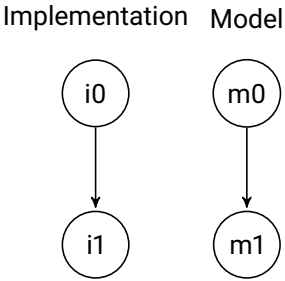
- Quviq QuickCheck
- PropEr

These tools are great for testing pure functions.

They have mechanisms to test stateful programs.

PBT state-machines or models.

A PBT model works like an oracle.



PBT models

Proprety-Based Testing (PBT) is a great testing methodology.

Successful tools widely used:

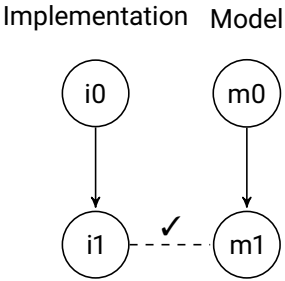
- Quviq QuickCheck
- PropEr

These tools are great for testing pure functions.

They have mechanisms to test stateful programs.

PBT state-machines or models.

A PBT model works like an oracle.



PBT models

Proprety-Based Testing (PBT) is a great testing methodology.

Successful tools widely used:

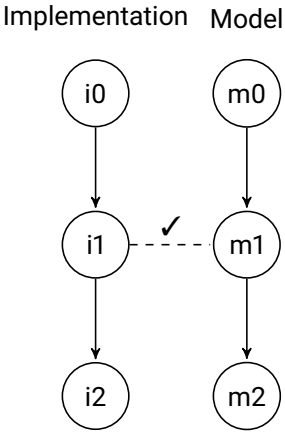
- Quviq QuickCheck
- PropEr

These tools are great for testing pure functions.

They have mechanisms to test stateful programs.

PBT state-machines or models.

A PBT model works like an oracle.



PBT models

Proprety-Based Testing (PBT) is a great testing methodology.

Successful tools widely used:

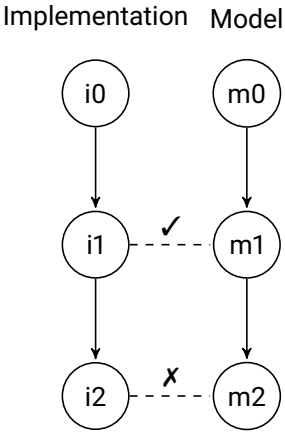
- Quviq QuickCheck
- PropEr

These tools are great for testing pure functions.

They have mechanisms to test stateful programs.

PBT state-machines or models.

A PBT model works like an oracle.



Problems with PBT models

Problems with PBT models

Despite their proven effectiveness:

- Very slow adoption

Problems with PBT models

Despite their proven effectiveness:

- Very slow adoption

Why?

1. Models are hard to reuse.
2. Bugs in models are hard to detect.
3. Errors are hard to understand.

Problems with PBT models

Despite their proven effectiveness:

- Very slow adoption

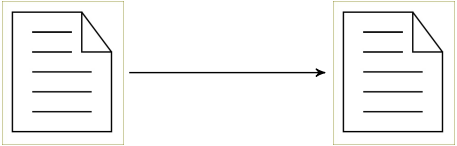
Why?

1. Models are hard to reuse.
2. Bugs in models are hard to detect.
3. Errors are hard to understand.

All these problems made models hard to write and maintain.

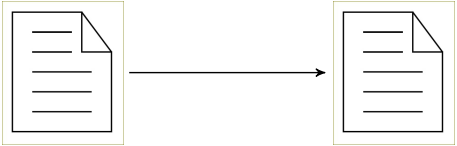
Our solution: Makina

Makina is a DSL for writing PBT models.



Our solution: Makina

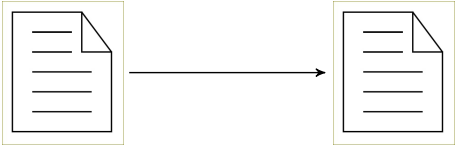
Makina is a DSL for writing PBT models.



1. Models are hard to reuse.
2. Bugs in models are hard to detect.
3. Errors are hard to understand.

Our solution: Makina

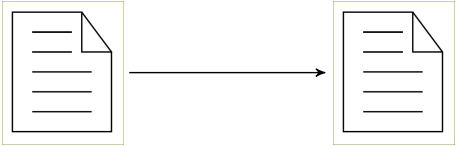
Makina is a DSL for writing PBT models.



1. Models are hard to reuse.
   - Modular reusable models.
2. Bugs in models are hard to detect.
3. Errors are hard to understand.

Our solution: Makina

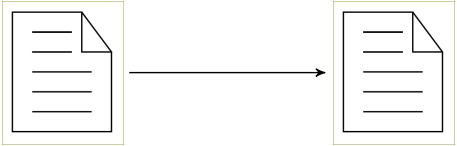
Makina is a DSL for writing PBT models.



1. Models are hard to reuse.
   - Modular reusable models.
2. Bugs in models are hard to detect.
   - Automatic type and specs generation.
3. Errors are hard to understand.

Our solution: Makina

Makina is a DSL for writing PBT models.



1. Models are hard to reuse.
   - Modular reusable models.
2. Bugs in models are hard to detect.
   - Automatic type and specs generation.
3. Errors are hard to understand.
   - Automatic runtime-checks generation.

Makina: The Language

Makina is implemented using Elixir macros.

```
defmodule Name do
  use Makina, [_option_]

  state [_attribute_]

  invariants [_invariants_]

  command _declaration_ do
    _command_body_
  end
end
```

Makina: The Language

Makina is implemented using Elixir macros.

```
defmodule Name do
  use Makina, [_option_]

  state [_attribute_]

  invariants [_invariants_]

  command _declaration_ do
    _command_body_
  end
end
```

*_option_*

- `extends: module()`
- `extends: [module()]`
- `implemented_by: module()`

Makina: The Language

Makina is implemented using Elixir macros.

```
defmodule Name do
  use Makina, [_option_]

  state [_attribute_]

  invariants [_invariants_]

  command _declaration_ do
    _command_body_
  end
end
```

*_option_*

- `extends: module()`
- `extends: [module()]`
- `implemented_by: module()`

*_attribute_*

- `name: expr`
- `name: expr type`

Makina: The Language

Makina is implemented using Elixir macros.

```
defmodule Name do
  use Makina, [_option_]

  state [_attribute_]

  invariants [_invariants_]

  command _declaration_ do
    _command_body_
  end
end
```

*_option_*

- `extends: module()`
- `extends: [module()]`
- `implemented_by: module()`

*_attribute_*

- `name: expr`
- `name: expr type`

*_declaration_*

- `name(arg1, ... , argN)`
- `name(arg1 type1, ... , argN typeN) return_type`

Makina: The Language

Makina is implemented using Elixir macros.

```
defmodule Name do
  use Makina, [_option_]

  state [_attribute_]

  invariants [_invariants_]

  command _declaration_ do
    _command_body_
  end
end
```

*_option_*

- `extends: module()`
- `extends: [module()]`
- `implemented_by: module()`

*_attribute_*

- `name: expr`
- `name: expr type`

*_declaration_*

- `name(arg1, ... , argN)`
- `name(arg1 type1, ... , argN typeN) return_type`

*_command_body_*

- `pre boolean()`
- `args generator()`
- `call return_type`
- `next [updates()]`
- `post boolean()`

Ethereum Blockchain

Why Ethereum?

- It is a complex system.

Ethereum Blockchain

Why Ethereum?

- It is a complex system.

API

`accounts!/0`	`accounts/0`	`block_number!/0`
`call_transaction!/4`	`call_transaction!/5`	`call_transaction/4`
`client_version!/0`	`client_version/0`	`compile_solidity!/1`
`deploy!/3`	`deploy!/4`	`deploy/3`
`estimate_gas!/4`	`estimate_gas!/5`	`estimate_gas/4`
`estimate_gas_cost!/4`	`estimate_gas_cost!/5`	`estimate_gas_cost/4`
`gas_cost!/1`	`gas_cost/1`	`gas_price!/0`
...	...	...

Ethereum Blockchain

Why Ethereum?

- It is a complex system.

API		
`accounts!/0`	`accounts/0`	`block_number!/0`
`call_transaction!/4`	`call_transaction!/5`	`call_transaction/4`
`client_version!/0`	`client_version/0`	`compile_solidity!/1`
`deploy!/3`	`deploy!/4`	`deploy/3`
`estimate_gas!/4`	`estimate_gas!/5`	`estimate_gas/4`
`estimate_gas_cost!/4`	`estimate_gas_cost!/5`	`estimate_gas_cost/4`
`gas_cost!/1`	`gas_cost/1`	`gas_price!/0`
...	...	...

The properties to test:

1. Mining blocks.
2. Account access.
3. Transactions between accounts.

Ethereum Blockchain

Why Ethereum?

- It is a complex system.

API		
`accounts!/0`	`accounts/0`	`block_number!/0`
`call_transaction!/4`	`call_transaction!/5`	`call_transaction/4`
`client_version!/0`	`client_version/0`	`compile_solidity!/1`
`deploy!/3`	`deploy!/4`	`deploy/3`
`estimate_gas!/4`	`estimate_gas!/5`	`estimate_gas/4`
`estimate_gas_cost!/4`	`estimate_gas_cost!/5`	`estimate_gas_cost/4`
`gas_cost!/1`	`gas_cost/1`	`gas_price!/0`
...	...	...

The properties to test:

1. Mining blocks.
2. Account access.
3. Transactions between accounts.

How *`Makina`* handles this complexity?

Mining blocks

Mining blocks

The API:

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

1. create module.

```elixir
defmodule Blocks do
```

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

```elixir
defmodule Blocks do
  use Makina
```

1. create module.
2. import `Makina`.

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

```elixir
defmodule Blocks do
  use Makina

  state height: 0
```

1. create module.
2. import `Makina`.
3. define state.

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

```elixir
defmodule Blocks do
  use Makina

  state height: 0

  invariants non_neg_height: height > 0
```

1. create module.
2. import `Makina`.
3. define state.
4. define invariants.

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

1. create module.
2. import *Makina*.
3. define state.
4. define invariants.
5. define commands.

```elixir
defmodule Blocks do
  use Makina

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
```

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

1. create module.
2. import *Makina*.
3. define state.
4. define invariants.
5. define commands.

```elixir
defmodule Blocks do
  use Makina

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    pre true
```

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

1. create module.
2. import *Makina*.
3. define state.
4. define invariants.
5. define commands.

```
defmodule Blocks do
  use Makina

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    pre true
    args []
```

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

1. create module.
2. import *Makina*.
3. define state.
4. define invariants.
5. define commands.

```elixir
defmodule Blocks do
  use Makina

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    pre true
    args []
    call Etherex.block_number
```

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

1. create module.
2. import *Makina*.
3. define state.
4. define invariants.
5. define commands.

```elixir
defmodule Blocks do
  use Makina

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    pre true
    args []
    call Etherex.block_number
    next []
```

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

1. create module.
2. import *Makina*.
3. define state.
4. define invariants.
5. define commands.

```
defmodule Blocks do
  use Makina

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    pre true
    args []
    call Etherex.block_number
    next []
    post height == result
```

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

1. create module.
2. import *Makina*.
3. define state.
4. define invariants.
5. define commands.

```
defmodule Blocks do
  use Makina

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    pre true
    args []
    call Etherex.block_number
    next []
    post height == result
  end
```

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

1. create module.
2. import *Makina*.
3. define state.
4. define invariants.
5. define commands.

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    post height == result
  end
```

Mining blocks

The API:

Command	Returns
`mine/0`	`:ok`
`block_number/0`	`integer()`

1. create module.
2. import *Makina*.
3. define state.
4. define invariants.
5. define commands.

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    post height == result
  end

  command mine() do
    next height: height + 1
  end
end
```

Running the test

```
$ mix test
```

```
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    post height == result
  end

  command mine() do
    next height: height + 1
  end
end
```

Running the test

```
$ mix test

Failed! After 1 tests.

Postcondition crashed:
** invariant "non_neg_height" check failed

block_number/0

Last state: %{height: 0}
```

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    post height == result
  end

  command mine() do
    next height: height + 1
  end
end
```

Running the test

```
$ mix test

Failed! After 1 tests.

Postcondition crashed:
** invariant "non_neg_height" check failed

block_number/0

Last state: %{height: 0}
```

This is a runtime check added by `Makina`!

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    post height == result
  end

  command mine() do
    next height: height + 1
  end
end
```

Fixing the model

```
$ mix test

Failed! After 1 tests.

Postcondition crashed:
** invariant "non_neg_height" check failed

block_number/0

Last state: %{height: 0}
```

This is a runtime check added by `Makina`!

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0

  invariants non_neg_height: height > 0

  command block_number() do
    post height == result
  end

  command mine() do
    next height: height + 1
  end
end
```

Fixing the model

```
$ mix test

Failed! After 1 tests.

Postcondition crashed:
** invariant "non_neg_height" check failed

block_number/0

Last state: %{height: 0}
```

This is a runtime check added by *Makina*!

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0

  invariants non_neg_height: height >= 0

  command block_number() do
    post height == result
  end

  command mine() do
    next height: height + 1
  end
end
```

Running the test

```
$ mix test
```

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0

  invariants non_neg_height: height >= 0

  command block_number() do
    post height == result
  end

  command mine() do
    next height: height + 1
  end
end
```

Running the test

```
$ mix test

..................................................
..................................................

OK, passed 100 tests

51.5 mine/0
48.5 block_number/0
```

```ruby
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0

  invariants non_neg_height: height >= 0

  command block_number() do
    post height == result
  end

  command mine() do
    next height: height + 1
  end
end
```

Adding type information

```
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0

  invariants non_neg_height: height >= 0

  command block_number() do
    post height == result
  end

  command mine() do
    next height: height + 1
  end
end
```

Adding type information

```
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0 :: integer()

  invariants non_neg_height: height >= 0

  command block_number() :: integer() do
    post height == result
  end

  command mine() :: :ok do
    next height: height + 1
  end
end
```

Adding type information

```
$ mix gradient

$
```

```
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0 :: integer()

  invariants non_neg_height: height >= 0

  command block_number() :: integer() do
    post height == result
  end

  command mine() :: :ok do
    next height: height + 1
  end
end
```

Adding type information

```
$ mix gradient

$
```

Something changes in `Etherex`...

```
defmodule Blocks do
  use Makina, implemented_by: Etherex

  state height: 0 :: integer()

  invariants non_neg_height: height >= 0

  command block_number() :: integer() do
    post height == result
  end

  command mine() :: :ok do
    next height: height + 1
  end
end
```

Adding type information

```
$ mix gradient

$
```

Something changes in `Etherex`...

```
$ mix gradient

The function call Etherex.block_number()
on line 8 is expected to have type integer()
but it has type
{:ok, quantity()} | {:error, error()}

$
```

```
1  defmodule Blocks do
2    use Makina, implemented_by: Etherex
3
4    state height: 0 :: integer()
5
6    invariants non_neg_height: height >= 0
7
8    command block_number() :: integer() do
9      post height == result
10   end
11
12   command mine() :: :ok do
13     next height: height + 1
14   end
15 end
```

Adding documentation

```
defmodule Blocks do
  use Makina, implemented_by: Etherex



  state height: 0 :: integer()

  invariants non_neg_height: height >= 0

  command block_number() :: integer() do

    post {:ok, height} == result
  end

  command mine() :: :ok do

    next height: height + 1
  end
end
```

Adding documentation

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  @moduledoc """
  Checks blocks are mined correctly.
  """
  state height: 0 :: integer()

  invariants non_neg_height: height >= 0

  command block_number() :: integer() do
    @moduledoc "Gets the block number."
    post {:ok, height} == result
  end

  command mine() :: :ok do
    @moduledoc "Mines a new block."
    next height: height + 1
  end
end
```

Adding documentation

```
iex> h Blocks
```

```
defmodule Blocks do
  use Makina, implemented_by: Etherex

  @moduledoc """
  Checks blocks are mined correctly.
  """
  state height: 0 :: integer()

  invariants non_neg_height: height >= 0

  command block_number() :: integer() do
    @moduledoc "Gets the block number."
    post {:ok, height} == result
  end

  command mine() :: :ok do
    @moduledoc "Mines a new block."
    next height: height + 1
  end
end
```

Adding documentation

```
iex> h Blocks

Contains a Makina model called Blocks.

Checks blocks are mined correctly.

## Commands

- mine
- block_number

## State attributes

- height

## Invariants

- non_neg_height
```

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  @moduledoc """
  Checks blocks are mined correctly.
  """
  state height: 0 :: integer()

  invariants non_neg_height: height >= 0

  command block_number() :: integer() do
    @moduledoc "Gets the block number."
    post {:ok, height} == result
  end

  command mine() :: :ok do
    @moduledoc "Mines a new block."
    next height: height + 1
  end
end
```

Adding documentation

```
iex> h Blocks.Command.Mine
```

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  @moduledoc """
  Checks blocks are mined correctly.
  """
  state height: 0 :: integer()

  invariants non_neg_height: height >= 0

  command block_number() :: integer() do
    @moduledoc "Gets the block number."
    post height == result
  end

  command mine() :: :ok do
    @moduledoc "Mines a new block."
    next height: height + 1
  end
end
```

Adding documentation

```
iex> h Blocks.Command.Mine

This module contains the functions necessary to
generate and execute the command mine.

Mines a new block.

## Definitions

- next
- call
- weight
- post
- args
- pre
```

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  @moduledoc """
  Checks blocks are mined correctly.
  """
  state height: 0 :: integer()

  invariants non_neg_height: height >= 0

  command block_number() :: integer() do
    @moduledoc "Gets the block number."
    post height == result
  end

  command mine() :: :ok do
    @moduledoc "Mines a new block."
    next height: height + 1
  end
end
```

Adding documentation

```
iex> h Blocks.Command.Mine.post
```

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  @moduledoc """
  Checks blocks are mined correctly.
  """
  state height: 0 :: integer()

  invariants non_neg_height: height >= 0

  command block_number() :: integer() do
    @moduledoc "Gets the block number."
    post height == result
  end

  command mine() :: :ok do
    @moduledoc "Mines a new block."
    next height: height + 1
  end
end
```

Adding documentation

```
iex> h Blocks.Command.Mine.post

This definition contains a predicate that should
be true after the execution of mine

## Available variables

### State

- state
- height

### Arguments

- arguments

### Result

- result
```

```elixir
defmodule Blocks do
  use Makina, implemented_by: Etherex

  @moduledoc """
  Checks blocks are mined correctly.
  """
  state height: 0 :: integer()

  invariants non_neg_height: height >= 0

  command block_number() :: integer() do
    @moduledoc "Gets the block number."
    post height == result
  end

  command mine() :: :ok do
    @moduledoc "Mines a new block."
    next height: height + 1
  end
end
```

Account access

The API:

Command	Returns
`balance/1`	`integer()`

Account access

The API:

Command	Returns
`balance/1`	`integer()`

1. create module.

```elixir
defmodule Accounts do

end
```

Account access

The API:

Command	Returns
`balance/1`	`integer()`

1. create module.
2. import *`Makina`*.

```elixir
defmodule Accounts do
  use Makina, implemented_by: Etherex

end
```

Account access

The API:

Command	Returns
`balance/1`	`integer()`

1. create module.
2. import *`Makina`*.
3. define state.

```elixir
defmodule Accounts do
  use Makina, implemented_by: Etherex

  @type balances() :: %{address() => integer()}

  state accounts: Etherex.accounts() :: [address()],
        balances: Etherex.balances() :: balances()

end
```

Account access

The API:

Command	Returns
`balance/1`	`integer()`

1. create module.
2. import *Makina*.
3. define state.
4. define commands.

```
defmodule Accounts do
  use Makina, implemented_by: Etherex

  @type balances() :: %{address() => integer()}

  state accounts: Etherex.accounts() :: [address()],
        balances: Etherex.balances() :: balances()

  command balance(account :: address()) :: integer() do
    pre accounts != []
    post balances[account] == result
  end
end
```

Running the test

```
$ mix test
```

```
defmodule Accounts do
  use Makina, implemented_by: Etherex

  @type balances() :: %{address() => integer()}

  state accounts: Etherex.accounts() :: [address()],
        balances: Etherex.balances() :: balances()

  command balance(account :: address()) :: integer() do
    pre accounts != []
    post balances[account] == result
  end
end
```

Running the test

```
$ mix test

** (Makina.Error) argument
‘account‘ missing in command
get_balance
```

```elixir
defmodule Accounts do
  use Makina, implemented_by: Etherex

  @type balances() :: %{address() => integer()}

  state accounts: Etherex.accounts() :: [address()],
        balances: Etherex.balances() :: balances()

  command balance(account :: address()) :: integer() do
    pre accounts != []
    post balances[account] == result
  end
end
```

Running the test

```
$ mix test

** (Makina.Error) argument
‘account‘ missing in command
get_balance
```

This is a runtime-check added by *Makina*!

```elixir
defmodule Accounts do
  use Makina, implemented_by: Etherex

  @type balances() :: %{address() => integer()}

  state accounts: Etherex.accounts() :: [address()],
        balances: Etherex.balances() :: balances()

  command balance(account :: address()) :: integer() do
    pre accounts != []
    post balances[account] == result
  end
end
```

Fixing the model

```
$ mix test

** (Makina.Error) argument
‘account‘ missing in command
get_balance
```

This is a runtime-check added by *Makina*!

```elixir
defmodule Accounts do
  use Makina, implemented_by: Etherex

  @type balances() :: %{address() => integer()}

  state accounts: Etherex.accounts() :: [address()],
        balances: Etherex.balances() :: balances()

  command balance(account :: address()) :: integer() do
    args account: oneof(accounts)
    pre accounts != []
    post balances[account] == result
  end
end
```

Running the test

```
$ mix test
```

```elixir
defmodule Accounts do
  use Makina, implemented_by: Etherex
  @type balances() :: %{address() => integer()}

  state accounts: Etherex.accounts() :: [address()],
        balances: Etherex.balances() :: balances()

  command balance(account :: address()) :: integer() do
    args account: oneof(accounts)
    pre accounts != []
    post balances[account] == result
  end
end
```

Running the test

```
$ mix test

.......................
.......................
.......................
.......................
OK, passed 100 tests

'100.0 get_balance/1
```

```elixir
defmodule Accounts do
  use Makina, implemented_by: Etherex
  @type balances() :: %{address() => integer()}

  state accounts: Etherex.accounts() :: [address()],
        balances: Etherex.balances() :: balances()

  command balance(account :: address()) :: integer() do
    args account: oneof(accounts)
    pre accounts != []
    post balances[account] == result
  end
end
```

Generating transactions

Generating transactions

The API to generate and check transactions:

Generating transactions

The API to generate and check transactions:

Command	Returns	Implemented
`mine/0`	`:ok`	✓
`block_number/0`	`integer()`	✓
`get_balance/1`	`integer()`	✓
`transfer/3`	`hash()`	

Generating transactions

The API to generate and check transactions:

Command	Returns	Implemented
`mine/0`	`:ok`	✓
`block_number/0`	`integer()`	✓
`get_balance/1`	`integer()`	✓
`transfer/3`	`hash()`	

We can compose *`Blocks`* and *`Accounts`*!

Generating transactions

The API to generate and check transactions:

Command	Returns	Implemented
`mine/0`	`:ok`	✓
`block_number/0`	`integer()`	✓
`get_balance/1`	`integer()`	✓
`transfer/3`	`hash()`	

We can compose *`Blocks`* and *`Accounts`*!

```
defmodule Transactions do
  use Makina,
    extends: [Blocks, Accounts],
    implemented_by: Etherex
end
```

Generating transactions

The API to generate and check transactions:

Command	Returns	Implemented
`mine/0`	`:ok`	✓
`block_number/0`	`integer()`	✓
`get_balance/1`	`integer()`	✓
`transfer/3`	`hash()`	

We can compose `Blocks` and `Accounts`!

```
defmodule Transactions do
  use Makina,
    extends: [Blocks, Accounts],
    implemented_by: Etherex
end
```

Generates a model `Transactions.Composed`.

Generating transactions

The API to generate and check transactions:

Command	Returns	Implemented
`mine/0`	`:ok`	✓
`block_number/0`	`integer()`	✓
`get_balance/1`	`integer()`	✓
`transfer/3`	`hash()`	

```
iex(1)> h Transactions.Composed
```

We can compose *Blocks* and *Accounts*!

```
defmodule Transactions do
  use Makina,
    extends: [Blocks, Accounts],
    implemented_by: Etherex
end
```

Generates a model *Transactions.Composed*.

Generating transactions

The API to generate and check transactions:

Command	Returns	Implemented
`mine/0`	`:ok`	✓
`block_number/0`	`integer()`	✓
`get_balance/1`	`integer()`	✓
`transfer/3`	`hash()`	

We can compose *Blocks* and *Accounts*!

```
defmodule Transactions do
  use Makina,
    extends: [Blocks, Accounts],
    implemented_by: Etherex
end
```

Generates a model *Transactions.Composed*.

```
iex(1)> h Transactions.Composed

# Transactions.Composed

## Commands

- mine stored
- get_balance
- block_number

## State attributes

- height
- balances
- accounts

## Invariants

- non_neg_height
```

Generating transactions

```
Transactions extends: Transactions.Composed.
```

```elixir
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

end
```

Generating transactions

```
Transactions extends: Transactions.Composed.
```

Command	Returns
`transfer/3`	`hash()`

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next balances: update(balances, from, to, value)
  end
end
```

Running the test

```
$ mix test
```

```elixir
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next balances: update(balances, from, to, value)
  end
end
```

Running the test

```
$ mix test

transfer("0xffcf8fdee72ac11",
         "0x90f8bf6a479f320",
         1)
block_number()

Postcondition failed.

block_number() -> {:ok, 1}

Last state: %{height: 0, ...}
```

```elixir
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next balances: update(balances, from, to, value)
  end
end
```

Fixing the model

```
$ mix test

transfer("0xffcf8fdee72ac11",
         "0x90f8bf6a479f320",
         1)
block_number()

Postcondition failed.

block_number() -> {:ok, 1}

Last state: %{height: 0, ...}
```

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,
         balances: update(balances, from, to, value)
  end
end
```

Running the test

```
$ mix test
```

```elixir
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,
         balances: update(balances, from, to, value)
  end
end
```

Running the test

```
$ mix test

transfer("0x90f8bf6a479f320",
         "0xffcf8fdee72ac11",
         1),
get_balance("0x90f8bf6a479f320")

Postcondition failed.

get_balance("0x90f8bf6a479f320")
-> {:ok, 979000}

Last state: %{
    balances: %{
        "0x90f8bf6a479f320" => 1000000
        .. }
    .. }
```

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,
         balances: update(balances, from, to, value)
  end
end
```

Fixing the model

To fix this error we need to extract the gas cost after producing a transaction.

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,
         balances: update(balances, from, to, value)
  end
end
```

Fixing the model

To fix this error we need to extract the gas cost after producing a transaction.

Model execution is performed in two phases:

1. Generation of the command sequence.
2. Real execution of the test.

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,
         balances: update(balances, from, to, value)
  end
end
```

Fixing the model

To fix this error we need to extract the gas cost after producing a transaction.

Model execution is performed in two phases:

1. Generation of the command sequence.
2. Real execution of the test.

PBT libraries solve this documenting:

- symbolic state: state of the model during phase 1.
- dynamic state: state of the model during phase 2.

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,
         balances: update(balances, from, to, value)
  end
end
```

Fixing the model

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]



  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,

         balances: update(balances, from, to, value)

  end

  command get_balance() do
    pre transactions == []
  end
end
```

Fixing the model

Makina makes the difference between symbolic and dynamic explicit.

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]



  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,

         balances: update(balances, from, to, value)

  end

  command get_balance() do
    pre transactions == []
  end
end
```

Fixing the model

Makina makes the difference between symbolic and dynamic explicit.

Provides two mechanisms to add information about symbolic state:

- `symbolic(t)` type.
- `symbolic(expr)` macro.

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]



  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,

         balances: update(balances, from, to, value)

  end

  command get_balance() do
    pre transactions == []
  end
end
```

Fixing the model

Makina makes the difference between symbolic and dynamic explicit.

Provides two mechanisms to add information about symbolic state:

- `symbolic(t)` type.
- `symbolic(expr)` macro.

Rules on symbolic state:

- An attribute with a symbolic type cannot be inspected in `next`.
- If we need to update a symbolic attribute we should use symbolic macro.

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]



  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,

         balances: update(balances, from, to, value)

  end

  command get_balance() do
    pre transactions == []
  end
end
```

Fixing the model

Makina makes the difference between symbolic and dynamic explicit.

Provides two mechanisms to add information about symbolic state:

- `symbolic(t)` type.
- `symbolic(expr)` macro.

Rules on symbolic state:

- An attribute with a symbolic type cannot be inspected in `next`.
- If we need to update a symbolic attribute we should use symbolic macro.

To fix our model we need

1. Add symbolic attributes to the state.

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]



  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,

         balances: update(balances, from, to, value)

  end

  command get_balance() do
    pre transactions == []
  end
end
```

Fixing the model

Makina makes the difference between symbolic and dynamic explicit.

Provides two mechanisms to add information about symbolic state:

- `symbolic(t)` type.
- `symbolic(expr)` macro.

Rules on symbolic state:

- An attribute with a symbolic type cannot be inspected in `next`.
- If we need to update a symbolic attribute we should use symbolic macro.

To fix our model we need

1. Add symbolic attributes to the state.

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  state transactions: [] :: [symbolic(hash())]
        balances: super() :: symbolic(balances)

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,

         balances: update(balances, from, to, value)

  end

  command get_balance() do
    pre transactions == []
  end
end
```

Fixing the model

Makina makes the difference between symbolic and dynamic explicit.

Provides two mechanisms to add information about symbolic state:

- `symbolic(t)` type.
- `symbolic(expr)` macro.

Rules on symbolic state:

- An attribute with a symbolic type cannot be inspected in `next`.
- If we need to update a symbolic attribute we should use symbolic macro.

To fix our model we need

1. Add symbolic attributes to the state.
2. Store and update symbolic attributes.

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  state transactions: [] :: [symbolic(hash())]
        balances: super() :: symbolic(balances)

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,

         balances: update(balances, from, to, value)

  end

  command get_balance() do
    pre transactions == []
  end
end
```

Fixing the model

Makina makes the difference between symbolic and dynamic explicit.

Provides two mechanisms to add information about symbolic state:

- `symbolic(t)` type.
- `symbolic(expr)` macro.

Rules on symbolic state:

- An attribute with a symbolic type cannot be inspected in `next`.
- If we need to update a symbolic attribute we should use symbolic macro.

To fix our model we need

1. Add symbolic attributes to the state.
2. Store and update symbolic attributes.

```
defmodule Transactions do
  use Makina,
    implemented_by: Etherex,
    extends: [Accounts, Blocks]

  state transactions: [] :: [symbolic(hash())]
        balances: super() :: symbolic(balances)

  command transfer(from, to, value) :: hash() do
    pre accounts != []
    args from: oneof(accounts),
         to: oneof(accounts),
         value: pos_integer()
    next height: height + 1,
         transactions: [result | transactions],
         balances: update(balances, from, to, value)
                        |> symbolic()
  end

  command get_balance() do
    pre transactions == []
  end
end
```

Fixing the model

```
defmodule Transactions.GasCost do
  use Makina, extends: Transactions

  end
end
```

We import `Transactions` model using `:extends`.

Fixing the model

We import `Transactions` model using `:extends`.

We add a command that gets the cost of a transaction.

```
defmodule Transactions.GasCost do
  use Makina, extends: Transactions

  end
end
```

Fixing the model

We import `Transactions` model using `:extends`.

We add a command that gets the cost of a transaction.

```
defmodule Transactions.GasCost do
  use Makina, extends: Transactions

  command gas_cost(hash :: hash())
      :: {address(), quantity()} do
    pre transactions != []
    args hash: oneof(transactions)
    next transactions: List.delete(transactions, hash),
         balances: update_gas(balances, result)
                   |> symbolic()
    end
  end
end
```

Running the test

```
$ mix test
```

```
defmodule Transactions.GasCost do
  use Makina, extends: Transactions

  command gas_cost(hash :: hash())
      :: {address(), quantity()} do
    pre transactions != []
    args hash: oneof(transactions)
    next transactions: List.delete(transactions, hash),
         balances: update_gas(balances, result)
                   |> symbolic()
    end
  end
end
```

Running the test

```
$ mix test

.........................
.........................
.........................
.........................

OK, passed 100 tests

'25.5 mine/0
'24.9 block_number/0
'23.6 transfer/3
'14.3 gas_cost/1
'11.8 get_balance/1
```

```
defmodule Transactions.GasCost do
  use Makina, extends: Transactions

  command gas_cost(hash :: hash())
      :: {address(), quantity()} do
    pre transactions != []
    args hash: oneof(transactions)
    next transactions: List.delete(transactions, hash),
         balances: update_gas(balances, result)
                   |> symbolic()
    end
  end
end
```

Results

Results

Problem on PBT models	`Makina` solution
Hard to reuse.	Modular and composable models.
Bugs are hard to detect.	Type and specs generation.
Generate cryptic errors.	Automatic runtime-checks.

Results

Problem on PBT models	`Makina` solution
Hard to reuse.	Modular and composable models.
Bugs are hard to detect.	Type and specs generation.
Generate cryptic errors.	Automatic runtime-checks.

Before `Makina`	After `Makina`
4 files 4513 lines	18 files 1692 lines

Links

Makina library:

- `https://gitlab.com/babel-upm/makina/makina/`

Etherex library:

- `https://gitlab.com/babel-upm/blockchain/etherex`

Slides and code examples:

- `https://github.com/lbueso/code_beam_2022`