



The COGENT Case for Property-Based Testing

Zilin Chen, Liam O'Connor, Gabriele Keller,
Gerwin Klein and [Gernot Heiser](#)

PLOS'17

www.data61.csiro.au

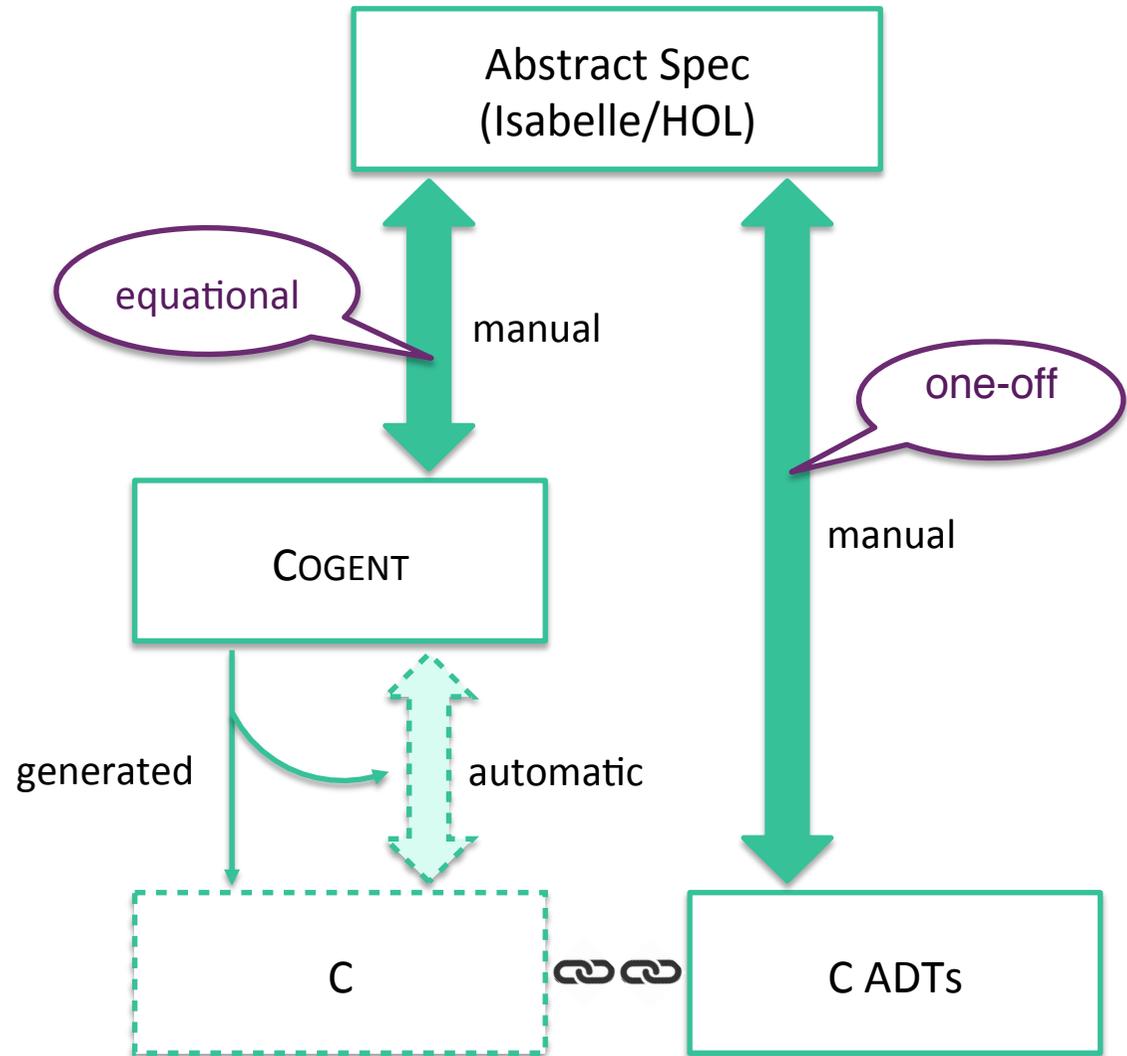


What Is COGENT?

- Reduces the cost of formally verifying systems code
- Restricted, purely functional language
- Uniqueness type system
- Case-studies: BilbyFs, ext2, F2FS, VFAT

O'Connor et al., ICFP'16

Amani et al., ASPLOS'16



Manual Proof Effort

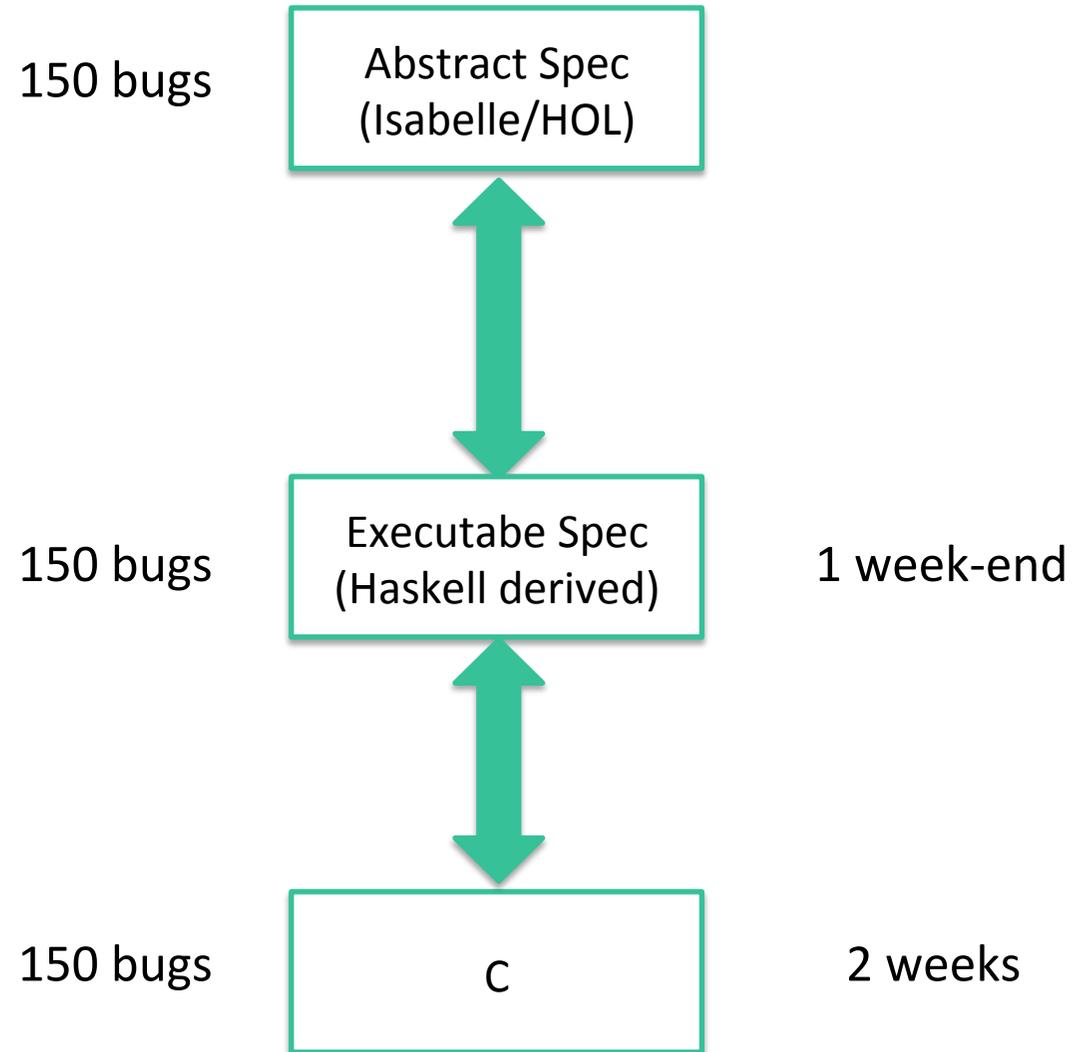


BilbyFs functions	Effort	Isabelle LOP	COGENT SLOC	Cost \$/SLOC	LOP/LOC
sync(), iget() and library	9.25 pm	13,000	1,350	150	10
sync()-specific	3.75 pm	5,700	300	260	19
iget()-specific	1 pm	1,800	200	100	9
seL4	12 py	180,000	8,700 C	350	20

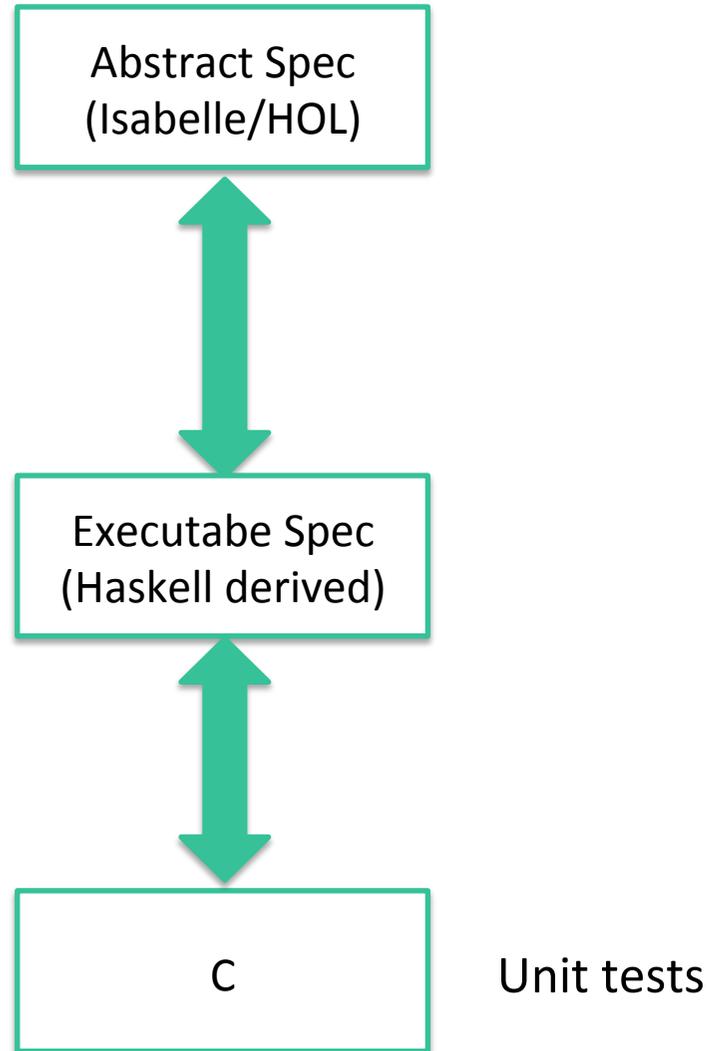
* BilbyFs: totally 4,200 lines of COGENT

Amani et al., ASPLOS'16

Original seL4



Present seL4: New Features



Assurance Strength

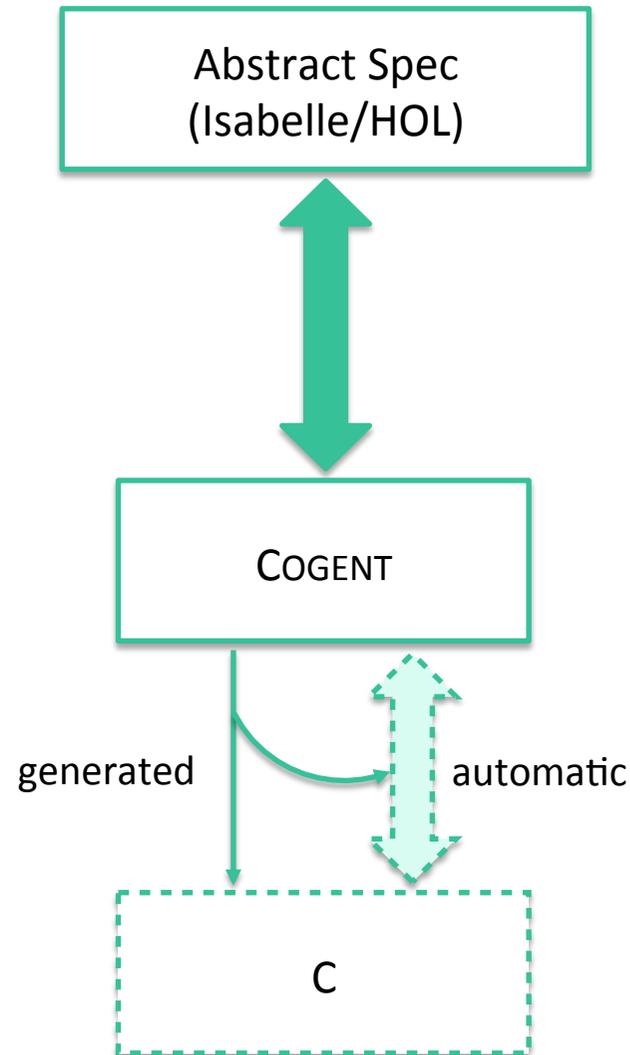
1. Safe language
2. Unit testing
3. Model checking
4. Model based testing
5. Functional proof

Can do in sequence

- Correct code easier to verify

Can stop anywhere

- Assurance – cost tradeoff



What Is Property-Based Testing (PBT)?



- `sort :: [Int] -> [Int]`
`sort = ...`

- Input: `[4,7,3,5,0]`
Expected: `[0,3,4,5,7]`

Unit testing:

- Feed in specified inputs
- Compare outputs to expected

- `prop_length: ∀xs. length xs == length (sort xs)`

`prop_elem: ∀xs, x. x∈xs ==> x∈(sort xs)`

`prop_sorted: ∀{xs | length xs >= 2}, i∈[0..length xs - 2].
(sort xs)!!i <= (sort xs)!!(i+1)`

`prop_lib: ∀xs. sort xs == List.sort xs`

Property-based testing:

- Generate random (but biased) inputs
- Run until violation found

COGENT: PBT-ing vs Verifying



- PBT overhead minimal if verifying anyway
 - Both need (formal) specification
 - Both need (formal) properties
- COGENT — total, deterministic, purely functional language
 - Good match to Isabelle & functional verification
- PBT: find bugs early (spec, properties, implementation)
- PBT: lightweight and agile alternative to formal proofs
- PBT and formal verification support each other!

What Do We Prove?



- Functional correctness:
a **refinement statement** from an **abstract specification**
- Definition (refinement):

Program C refines A , if $\llbracket C \rrbracket \subseteq \llbracket A \rrbracket$.

- **Refinement relation** R that relates abstract and concrete states
- Data refinement (informally):

$$\begin{array}{ccc} I_A & \xrightarrow{P(A)} & F_A \\ R \left\{ & & \right\} R \\ I_C & \xrightarrow{P(C)} & F_C \end{array}$$

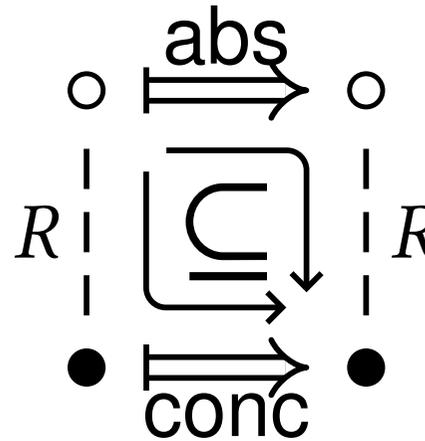
de Roever and Engelhardt, 1998

Proving Refinement



- To prove data refinement (by simulation):

$$R; \llbracket \text{conc} \rrbracket \subseteq \llbracket \text{abs} \rrbracket; R$$



- COGENT: purely functional, deterministic, total
- Refinement statement:

$$\text{abs} : X_a \rightarrow Y_a \quad \text{and} \quad \text{conc} : X_c \rightarrow Y_c$$

$$R_X i_a i_c \Rightarrow \exists o_a \in \text{abs } i_a. R_Y o_a (\text{conc } i_c) \quad (1)$$

$$\text{corres } R a c \triangleq \exists o \in a. R o c$$

$$\boxed{R_X i_a i_c \Rightarrow \text{corres } R_Y (\text{abs } i_a) (\text{conc } i_c)} \quad (2)$$

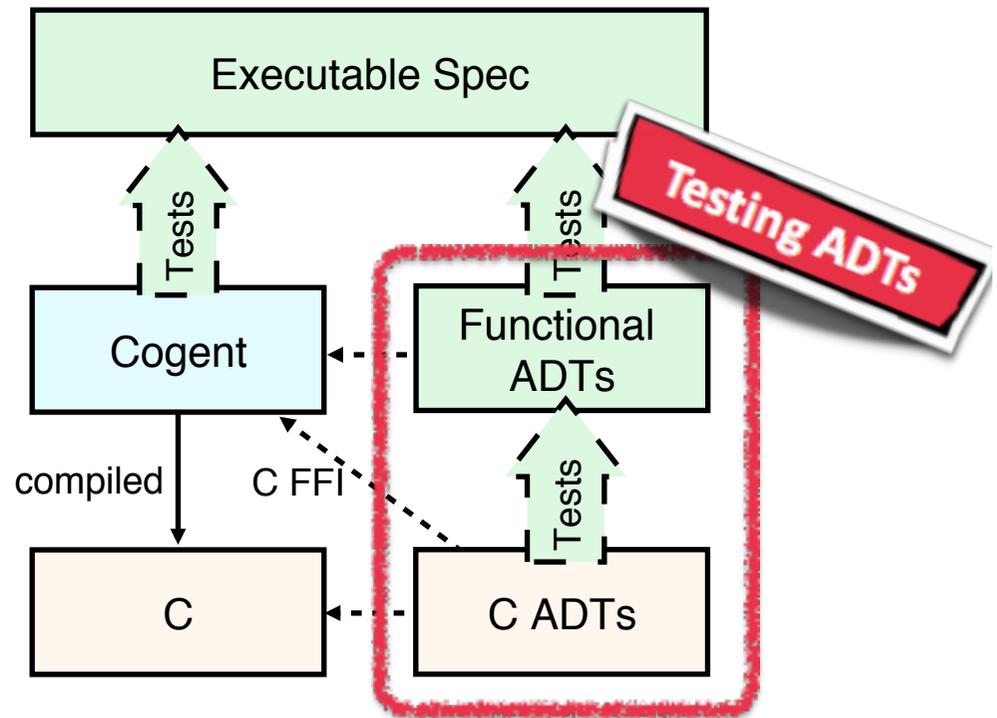
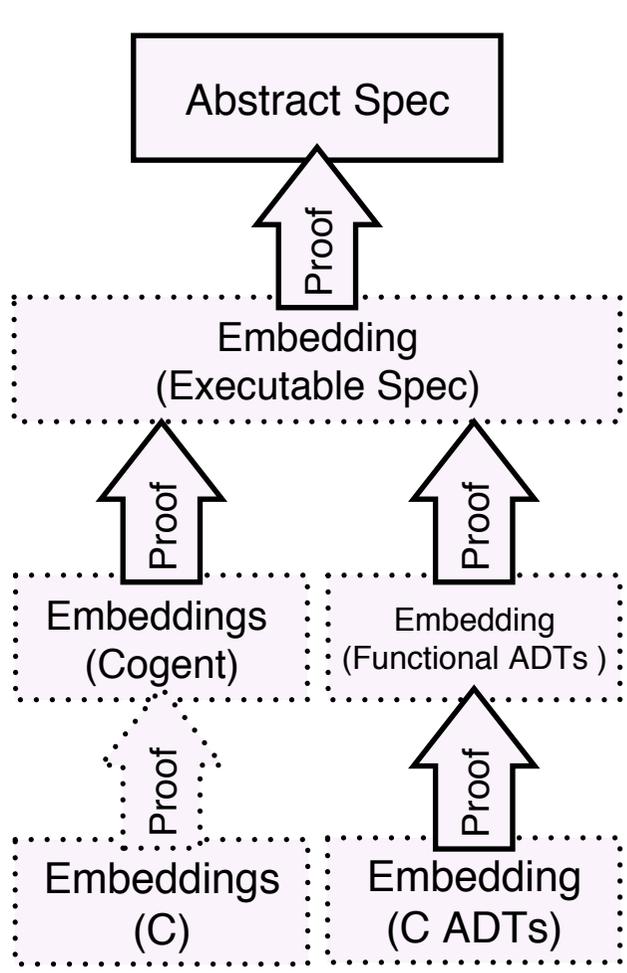
How To Test Refinement



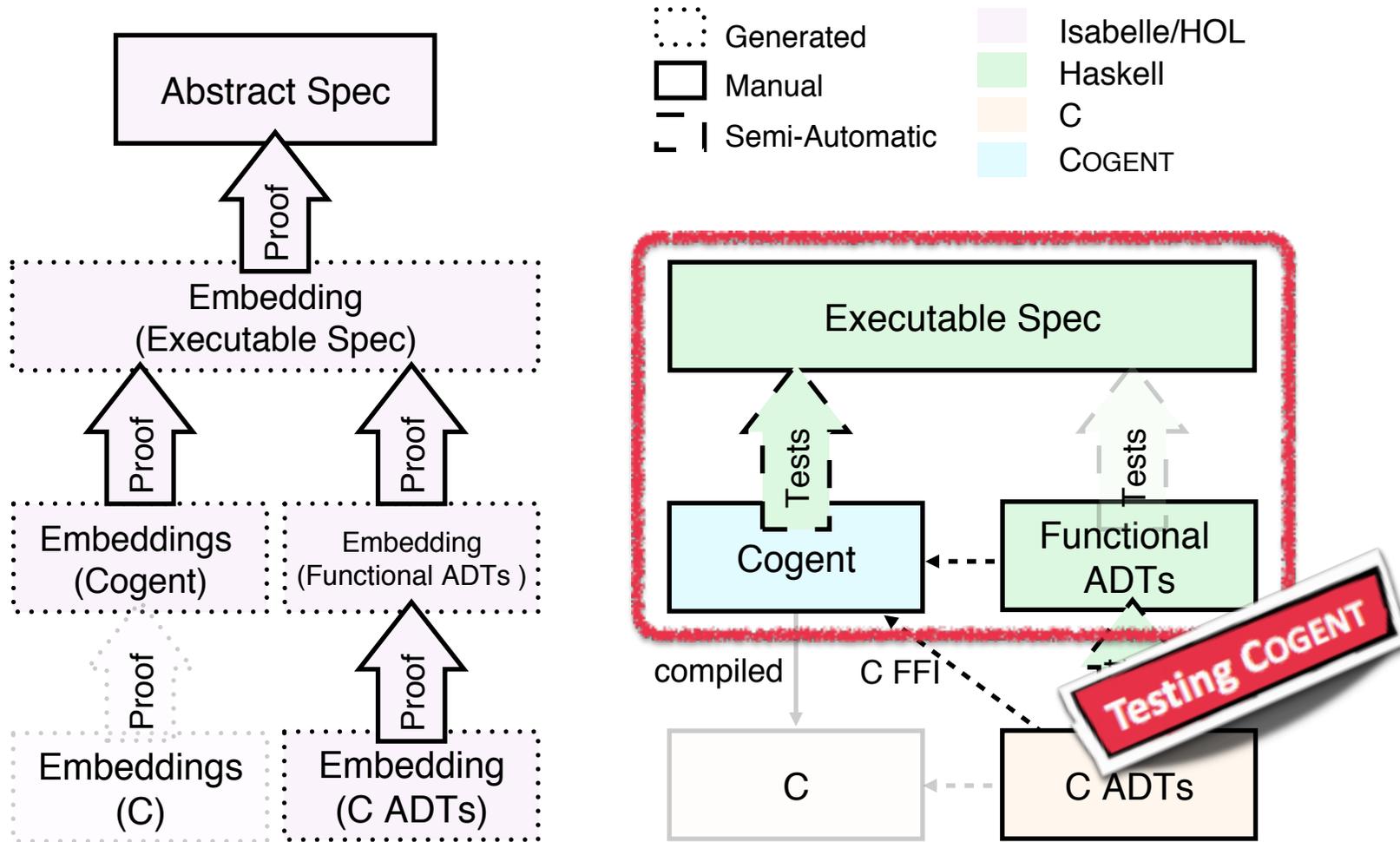
$$R_X i_a i_c \Rightarrow \text{corres } R_Y (\text{abs } i_a) (\text{conc } i_c)$$

- Encode these properties as machine-testable specifications
- QuickCheck library in Haskell Claessen and Hughes, ICFP'00
 - test data generation — control distribution, satisfy invariants
 - combinators in the specification language: `forall`, `(==>)`, `(.&.)`, `(.&&.)`, `(.||.)`, etc.
 - counter-example shrinking

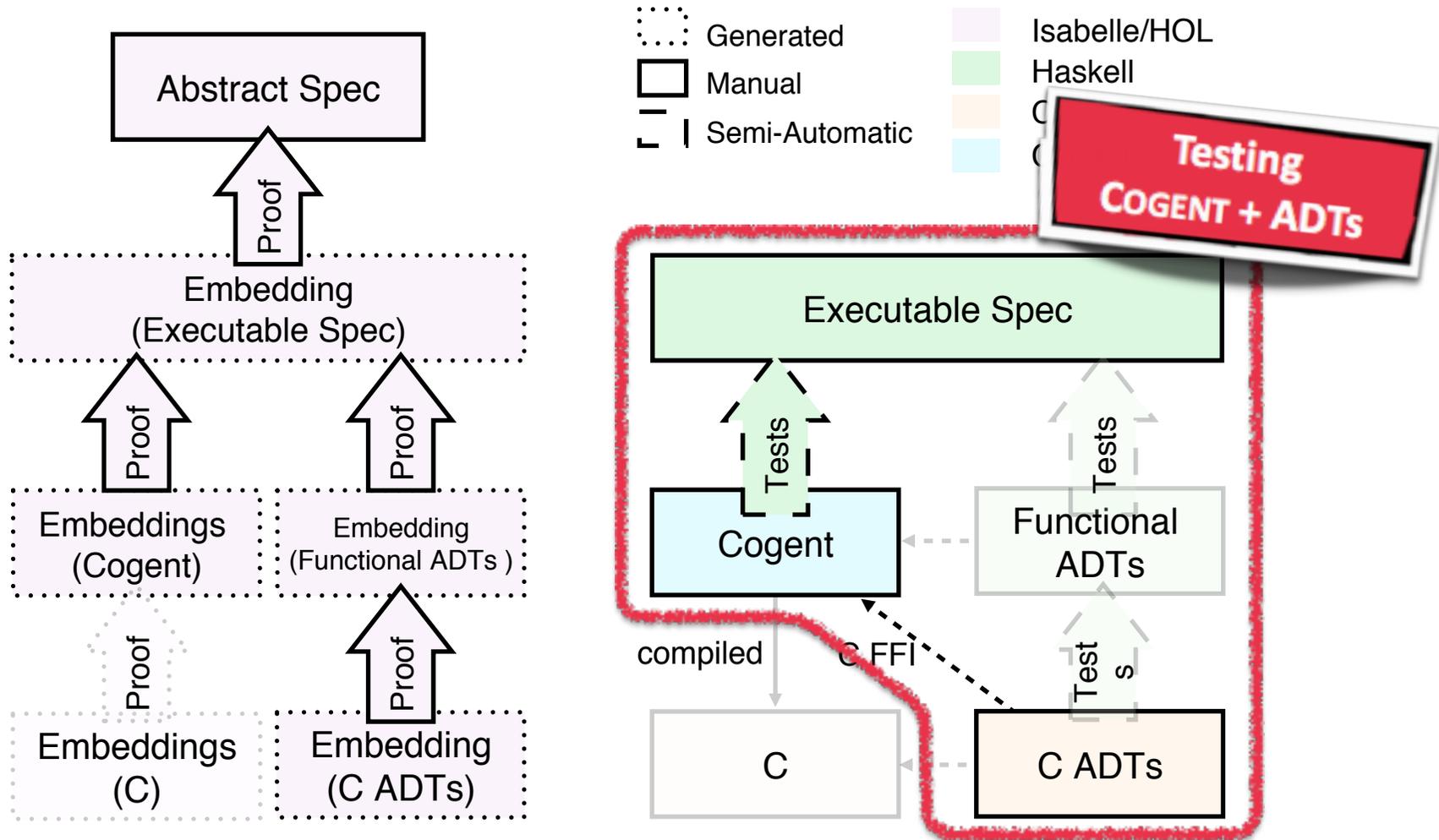
QuickCheck Architecture



Modular Testing



Modular Testing



Example



```
data MountState
data FsmState
hs_fsm_init :: (MountState, FsmState)
             -> Cogent_monad (Either ErrCode FsmState)
```

```
fsm_init : (SysState, MountState!, FsmState take (..))
          -> RR SysState FsmState (ErrCode, FsmState take(..))
```

```
cogent_fsm_init :: Ct21 -> IO Ct24
foreign import ccall unsafe "ffi_fsm_init"
c_fsm_init :: Ptr FFI.Ct21 -> IO (Ptr FFI.Ct24)
```

```
t24* ffi_fsm_init (t21* a1);
```

```
t24 fsm_init (t21 a1);
```

```
data Ct21
data Ct24
```

Example



```
hs_fsm_init :: (MountState, FsmState)
             -> Cogent_monad (Either ErrCode FsmState)
```

```
a2c_I :: (MountState, FsmState) -> IO Ct21
```

```
rel_0 :: Either ErrCode FsmState -> Ct24 -> IO Bool
```

```
gen_MountState :: Gen MountState
gen_FsmState   :: Gen FsmState
```

```
$ghci > quickCheck prop_fsm_init_corres
+++ OK, passed 100 tests.
```

```
prop_fsm_init_corres = monadicIO $
  forAllM gen_MountState $ \mount_st ->
  forAllM gen_FsmState   $ \fsm_st   -> run $ do
    let ia = (mount_st, fsm_st)
        ic ← a2c_I ia
        oa ← return $ hs_fsm_init (mount_st, fsm_st)
        oc ← cogent_fsm_init ic
    corresM rel_0 oa oc
```

```
cogent_fsm_init :: Ct21 -> IO Ct24
```

Refines

More In The Paper



The COGENT Case for Property-Based Testing

Zilin Chen Liam O'Connor Gabriele Keller Gerwin Klein Gernot Heiser

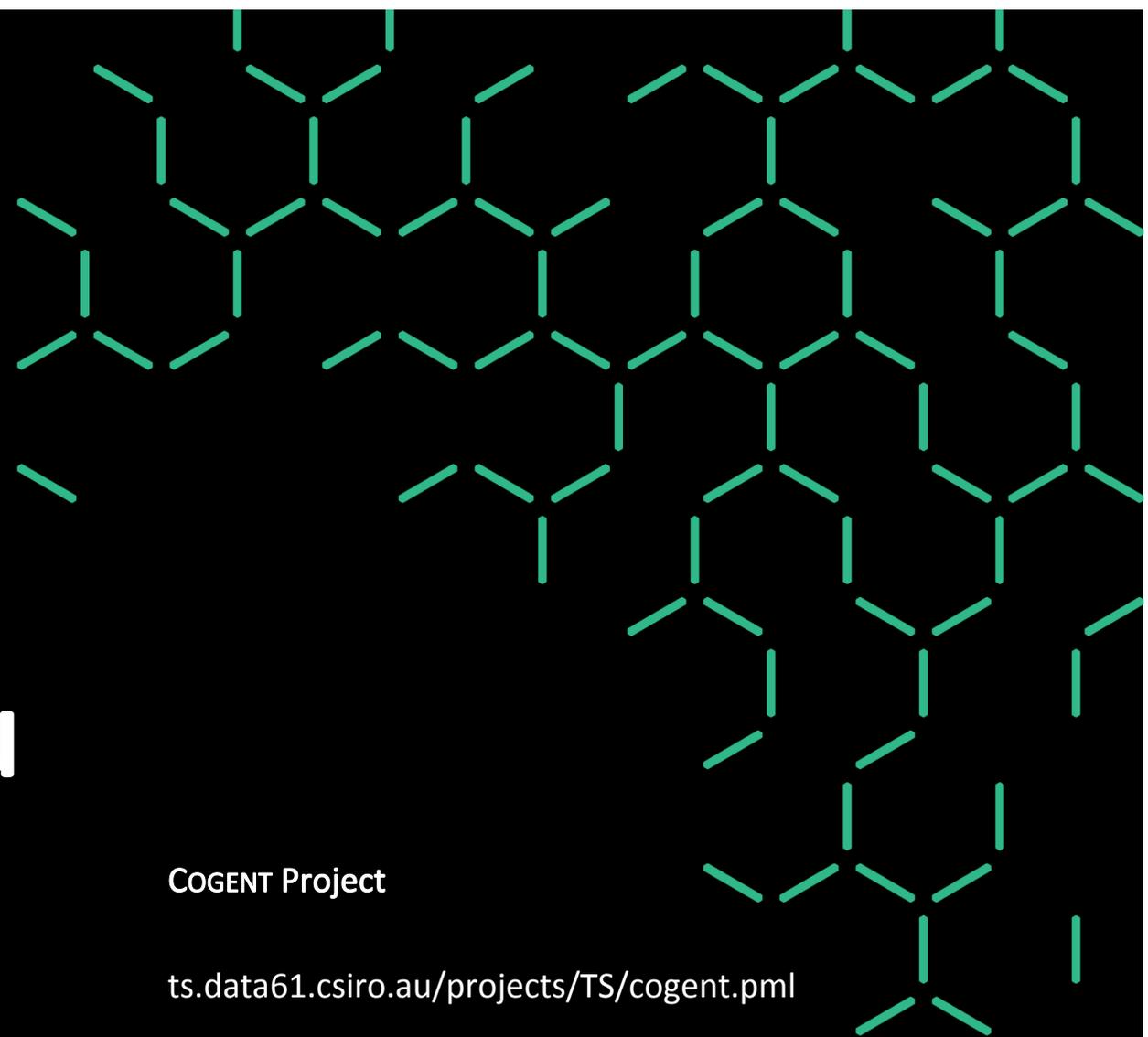
Data61 and UNSW, Australia

`first.last@data61.csiro.au`

Future Work



- More automation
 - Glue code between Haskell and C
 - Refinement statements $R_X i_a i_c \Rightarrow \text{corres } R_Y (\text{abs } i_a) (\text{conc } i_c)$
 - Refinement relations R_X, R_Y with domain-specific knowledge
 - Isabelle theorems
 - Test results vs. theorem proving in Isabelle
- Test data generators and shrinking algorithms
 - David R. Maclver, 2016
 - Jacob Stanley, 2017
- Testing kernel modules
- Full case-studies



Thank you

Trustworthy Systems

COGENT Project

Gernot Heiser

e gernot.heiser@data61.csiro.au

w <http://trustworthy.systems>

ts.data61.csiro.au/projects/TS/cogent.pml

www.data61.csiro.au

