User Documentation for StatWhy v.1.0

Yusuke Kawamoto[1], Kentaro Kobayashi[1,2], and Kohei Suenaga[3] *

[1] AIST, Tokyo, Japan
[2] University of Tsukuba, Ibaraki, Japan
[3] Kyoto University, Kyoto, Japan

Abstract. StatWhy is a software tool for automatically checking whether a programmer has properly specified the requirements for the statistical methods. In this documentation, we first present how to install and use the StatWhy tool. Then we demonstrate how the tool can avoid common errors in the application of a variety of hypothesis testing methods.

1 Foreword

StatWhy is a software tool for automatically checking whether a programmer has properly specified the requirements for the statistical methods.

In Section 2, we first present how to install the StatWhy tool. In Section 4, we show an illustrating example to see how a tool user can verify a statistical program. In Section 3, we present how to execute the StatWhy tool. In Section 5, we demonstrate how we can prevent common errors in the application of a variety of hypothesis testing methods by using StatWhy.

1.1 Availability

The source code of the StatWhy tool is available as a zip file "statwhy-1.0.0.zip." We plan to publish the source code before the artifact submission deadline (June 17th, 2024).

1.2 Contact

Report any bugs and requests to yusuke.kawamoto@aist.go.jp by email.

1.3 Acknowledgments.

The authors are supported by JSPS KAKENHI Grant Number JP24K02924, Japan. Yusuke Kawamoto is supported by JST, PRESTO Grant Number JPMJPR2022, Japan, and by JSPS KAKENHI Grant Number JP21K12028, Japan. Kohei Suenaga is supported by JST CREST Grant Number JPMJCR2012, Japan.

* The authors are listed in the alphabetical order.

2 Installation

In this section, we explain how to install StatWhy and our extension of the Cameleer tool [5]. This tutorial is intended for macOS because we tested StatWhy only on it.

2.1 Installing OCaml

First, we need to install OCaml via the official package manager opam.

1. Install opam
   To install opam with Homebrew:

   ```
   $ brew install opam
   ```

   Alternatively, if you use MacPorts:

   ```
   $ port install opam
   ```

   After the installation of opam, you need to initialize it:

   ```
   $ opam init -y
   $ eval $(opam env)
   ```

2. Install OCaml 5.0
   We use OCaml 5.0 because on the newer version, 5.1, we encountered a dynamic link issue with Cameleer. To install OCaml 5.0, do the following:

   ```
   $ opam switch create 5.0.0
   $ eval $(opam env)
   ```

2.2 Installing StatWhy and Cameleer

Download the zip file `statwhy-1.0.0.zip` for the source code of StatWhy, including an extension of Cameleer. Install StatWhy and Cameleer by running:

```
$ unzip statwhy-cameleer-1.0.0.zip
$ cd cameleer
$ opam pin add .
```

This will install Cameleer, StatWhy (included in "cameleer/src"), and their dependencies. In the installation of Cameleer, the Why3 platform [2] is automatically installed.

We recommend installing SMT solvers Alt-Ergo, CVC4, and Z3. To install them, do the following:

```
# Install Alt-Ergo
$ opam install alt-ergo

# Install CVC5
```

```
$ brew tap cvc5/homebrew-cvc5
$ brew install cvc5

# Install Z3
$ brew install z3
```

After installing these solvers, run the following:

```
 $ why3 config detect
```

2.3 Installing Pyml, Scipy

These modules are necessary when you execute the programs in "platform/examples."

Install Pyml by running:

```
$ opam install pyml
```

Install Scipy by running:

```
$ python3 -m pip install scipy
```

We have tested our codes with Python 3.9.6

3 Usage

You can verify OCaml codes via Cameleer by running the following:

```
$ statwhy <file-to-be-verified>.ml
```

If you want to verify WhyML codes:

```
$ statwhy <file-to-be-verified>.mlw
```

Note that in both cases, you need our extension of Cameleer to load StatWhy.

4 Getting Started

We show an example of an OCaml program annotated with preconditions and postconditions that we want to verify using the StatWhy tool. In the code below, the function `example1` conducts a two-sided t-test for a mean of a population, given a dataset `d` as input. The function `ex` applies the function `example1` to an actual dataset `y`. The function `res` executes the function `ex`.

platform/examples/example1.ml

```
open StatEff.CameleerBHL

module Example1 = struct
```

```
  open StatEff.Ttest

  let t_mu : term = Real (100, "mu")
  let t_mu0 : term = RealConst 1.0
  let fmlA_l : formula = Atom (Pred ("<", [ t_mu; t_mu0 ]))
  let fmlA_u : formula = Atom (Pred (">", [ t_mu; t_mu0 ]))
  let fmlA : formula = Atom (Pred ("=!", [ t_mu; t_mu0 ]))

  let example1 (d : float list dataset) : float =
    exec_ttest_1samp t_mu 1.0 d Two
  (*@ p = example1 d
    requires
      is_empty (!st) /\
      is_normal (ppl d) /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u
    ensures
      p = compose_pvs fmlA !st &&
      (World !st interp) |= StatB p fmlA
  *)

  let y =
    [
      2.40104652638315974;
      1.03084261391788212;
      0.916057617917251488;
      0.835577550392823665;
      1.35012678223086802;
      1.49031154485472528;
      1.55573484682964347;
      1.89213839194442013;
      0.810443568544103;
      1.1821574446281029;
    ]

  let ex () = example1 y
  (*@ requires
      is_empty !st /\
      is_normal (ppl y) /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u *)

  let[@run] res =
    let res, _ = run ex in
    Format.printf "p-value : %f\n" res
end
```

We explain more details on the function `example1` as follows. The specification for `example1` is written in the GOSPEL specification language [1][4]. The comment sections starting with `(*@` denote the specifications. More information on the syntax of GOSPEL can be found on the following webpage: https://ocaml-gospel.github.io/gospel/language/syntax.

Now we explain the details of the function `example1` as follows.

```
let example1 (d : float list dataset) : float =
  exec_ttest_1samp t_mu 1.0 d Two
(*@ p = example1 d
    requires
      is_empty (!st) /\
      is_normal (ppl d) /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u
    ensures
      p = compose_pvs fmlA !st &&
      (World !st interp) |= StatB p fmlA
*)
```

`p = example1 d` assigns the result of `example1 d` to the name `p`, allowing us to refer it as `p` throughout the specification. The `requires` and the `ensures` clauses specify the precondition and postcondition of `example1 d`, respectively. Concretely, the precondition consists of:

- `is_empty (!st)` represents that the history `st` of hypothesis testing is empty; i.e., no hypothesis test have performed so far;
- `is_normal (ppl d)` represents that the population `(ppl d)` which a dataset `d` was sampled from follows a normal distribution;
- `(World (!st) interp) |= Possible fmlA_l` represents that the analyst has a prior belief that the lower-tail `fmlA_l` (i.e., the hypothesis that the population mean is less than 1.0) can be true;
- `(World (!st) interp) |= Possible fmlA_u` represents that the analyst has a prior belief that the upper-tail `fmlA_u` (i.e., the hypothesis that the population mean is greater than 1.0) can be true.

The postcondition consists of:

- `p = compose_pvs fmlA !st` represents that `p` is the p-value obtained by the hypothesis test with the alternative hypothesis `fmlA`;
- `(World !st interp) |= StatB p fmlA` represents that the analyst obtains a belief on the alternative hypothesis `fmlA` with the p-value `p`. This logical formula (`StatB p fmlA`) employs a statistical belief modality `StatB` introduced in belief Hoare logic (BHL) [3,4].

For the instructions of verifying this code, see Section 5.1.

[4] https://github.com/ocaml-gospel/gospel

4.1 Remarks on Modules and Expressions

- We need to open `CameleerBHL` and `Ttest` (or any hypothesis testing modules) explicitly
- We add the attribute `[@run]` to ignore `let` expressions that are used exclusively for the execution and should not be verified by StatWhy. This attribute is useful when we want to avoid verifying a particular function. In the above example code, StatWhy does not verify the function `res`.
- We cannot use the "and" pattern `_` and the "unit" pattern `()` in top-level definitions. The current version of Cameleer does not support these patterns in top-level definitions. For instance, we cannot use the following expression "`let[@run] _ = ...`."

5 Examples of Analyses Using StatWhy

We show a variety of examples on how StatWhy can be used to avoid common errors in hypothesis testing.

- In Section 5.1, we demonstrate how to use StatWhy in an example of a one-sample t-test.
- In Section 5.2, we show how StatWhy can check whether appropriate kinds of t-test are applied to different situations.
- In Section 5.3, we explain how StatWhy can verify a program for the Bonferroni correction in multiple comparison and address p-value hacking problems.
- In Section 5.4, we show how StatWhy can check the programs with ANOVA and other hypothesis tests under different requirements.
- In Section 5.5, we present an experimental evaluation of the execution times of StatWhy using a variety of hypothesis testing programs.

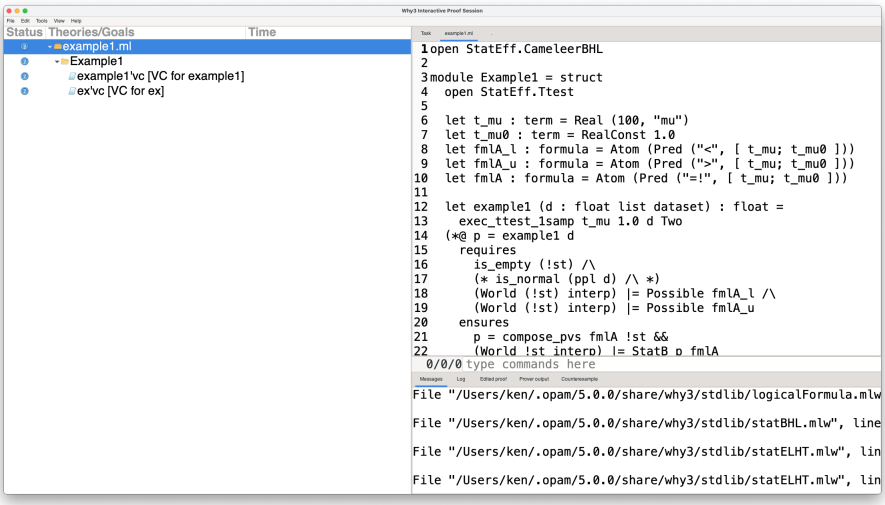
5.1 Example 1. Simple t-test (one-sample t-test)

In this example, we demonstrate how to use StatWhy by verifying a program conducting the *t-test for a mean of a population*.

To verify `Examples1` in `platform/examples/example1.ml`, execute the following command:

```
$ cd platform/examples
$ cameleer example1.ml
```

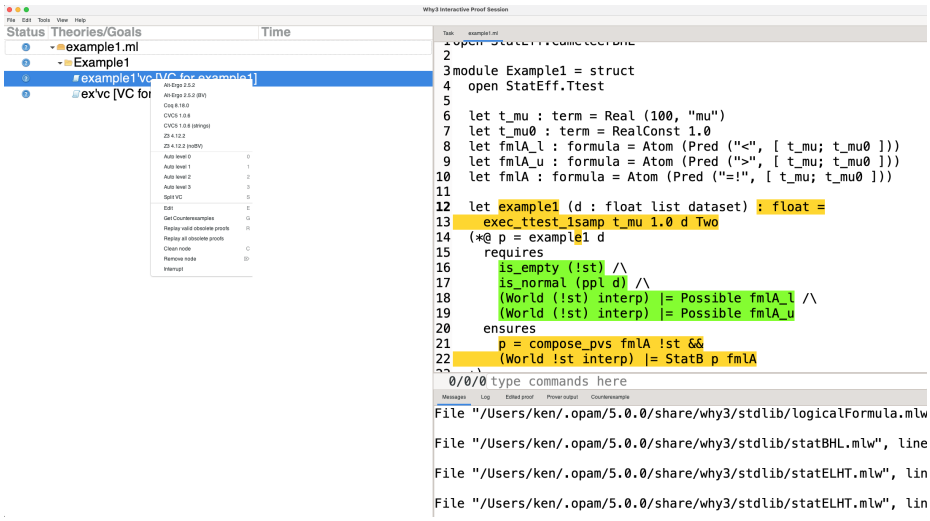
This command transforms the OCaml code into WhyML code, generates the verification conditions (VC), and opens the code in Why3 IDE.



The Why3 IDE screen displays the code to be verified in the right panel and shows the VCs generated from the code in the left panel. Why3 generates two VCs from 'example1.ml' in this case, `example1'vc` and `ex'vc`.

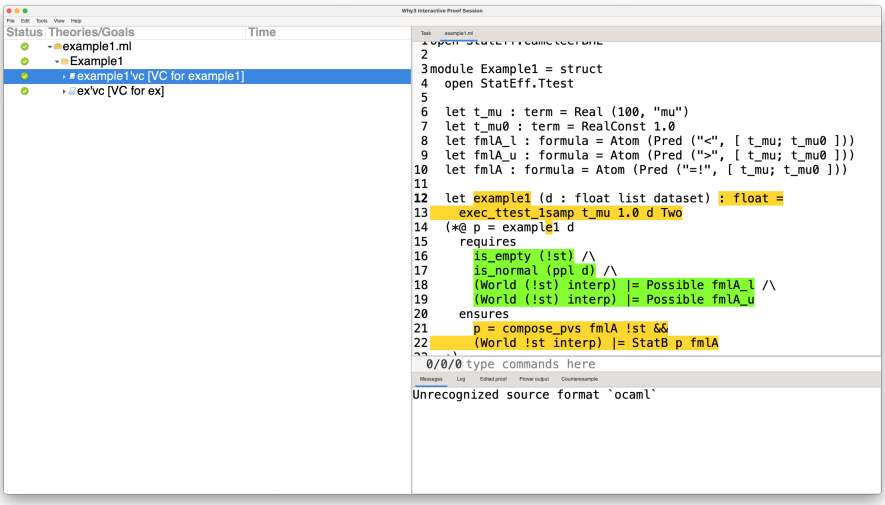
`example1'vc` is a VC for our hypothesis testing program `example1`; `ex'vc` is a VC for applying actual data `y`.

To prove each VC, right-click on the VC, then click "Auto level 3".

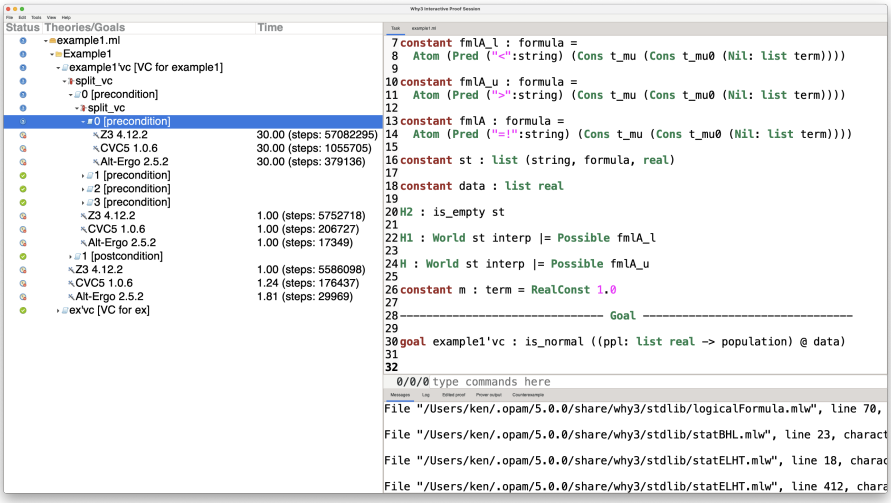


"Auto level 3" attempts to verify VCs by performing several transformations and forwarding them to available SMT solvers.

If one prover succeeds in verifying a VC, a check-mark will appear to the left of the VC, indicating that the VC is correct:



If Why3 fails to validate the VC, we can see which conditions do not hold by looking at the results of each sub-condition generated by "Auto level 3". The following figure, for example, shows a case where the precondition `is_normal (ppl d)` is commented out.



Finally, we remark that we can actually run the OCaml program `example1.ml` as follows:

```
$ cd platform/examples
$ dune exec -- ./example1.exe
```

If the dependencies are installed correctly, we obtain the following output:

```
$ dune exec -- ./example1.exe
p-value : 0.059935
```

5.2 Example 2. Several Variants of *t*-tests

StatWhy can distinguish between different preconditions for each hypothesis test command and remind users to make such conditions explicit. In this example, we will consider several variants of t-tests, such as paired/non-paired t-tests and t-tests in the presence of populations with equal/unequal variance.

Paired *t*-test vs Non-Paired *t*-test We use the *paired t-test* when there is a pairing or matching between the two samples. On the other hand, the *non-paired (or independent samples) t-test* is applied when two datasets are sampled from the population independently. StatWhy can check which of the tests should be applied to a situation that satisfies the preconditon.

Paired *t*-test The specification of the paired t-test command `exec_ttest_paired` in StatWhy is as follows:

```
val exec_ttest_paired (r1 : term) (r2 : term) (y : dataset (list real,
    list real)) (alt : alternative) : real
writes { st }
requires {
  match y with
  | (y1, y2) ->
    is_normal (ppl y1) /\ is_normal (ppl y2) /\ paired y1 y2 /\
      (exists r1':real_invisible. exists s:int. r1 = Real s r1') /\
      (exists r2':real_invisible. exists s:int. r2 = Real s r2') /\
      match alt with
      | Two ->
          (World !st interp) |= Possible (Atom (Pred "<" (Cons r1 (Cons
              r2 Nil)))) /\
          (World !st interp) |= Possible (Atom (Pred ">" (Cons r1 (Cons
              r2 Nil))))
      | Up ->
          (World !st interp) |= Not (Possible (Atom (Pred "<" (Cons r1 (
              Cons r2 Nil))))) /\
          (World !st interp) |= Possible (Atom (Pred ">" (Cons r1 (Cons
              r2 Nil))))
      | Low ->
          (World !st interp) |= Possible (Atom (Pred "<" (Cons r1 (Cons
              r2 Nil)))) /\
          (World !st interp) |= Not (Possible (Atom (Pred ">" (Cons r1 (
              Cons r2 Nil)))))
      end
  end
```

```
}
ensures {
  let pv = pvalue () in
  result = pv /\
  let h = match alt with
    | Two -> Pred "=!" (Cons r1 (Cons r2 Nil))
    | Up -> Pred ">" (Cons r1 (Cons r2 Nil))
    | Low -> Pred "<" (Cons r1 (Cons r2 Nil))
  end in !st = !(old st) ++ Cons ("ttest_ind", Atom h, pv) Nil
}
```

`exec_ttest_paired` has four arguments. `r1` and `r2` are the terms that represent population means, `y` is the pair of samples, and `alt` denotes what type of alternative hypothesis is; `Two` is for two-tailed tests, `Up` is for upper-tailed tests, and `Low` is for lower-tailed tests.

The precondition for the test is in the `requires` section. `is_normal (ppl y1)` and `is_normal (ppl y2)` assert that the populations of `y1` and `y2` are normally distributed. `paired y1 y2` means that two samples `y1` and `y2` are *not* obtained in pairs — they are independent. `(exists r1':real_invisible. exists s:int. r1 = Real s r1')` checks whether the given terms represent real numbers; we call them `real_invisible` since variables such as population means are *invisible* to us. Since StatWhy's hypothesis formulas have no type, StatWhy can accept terms that do not represent population means.

The formula `(World !st interp) |= Possible (Atom (Pred "<" (Cons r1 (Cons r2 Nil))))` denotes that we have a prior belief on alternative hypotheses (i.e., $\mathbf{P}$ (`r1` $<$ `r2`)) in the world `(World !st interp)`. We have implemented epistemic logic in Why3, and `|=` is used to represent the satisfaction of a modal formula in a world. The left-hand side of `|=` is a possible world, a tuple of `store` and `interpretation`. A `store` is a list of the results from each hypothesis test that has been performed so far. We use stores to interpret logical formulas about p-values. An `interpretation` is a function that maps `real_invisible` to `real`.

The `ensures` clause specifies the postcondition for the test. The expression `!st = !(old st) ++ Cons ("ttest_ind", Atom h, pv) Nil` stores the resulting p-value `pv` and the alternative hypothesis (`Atom h`) in the store `st`.

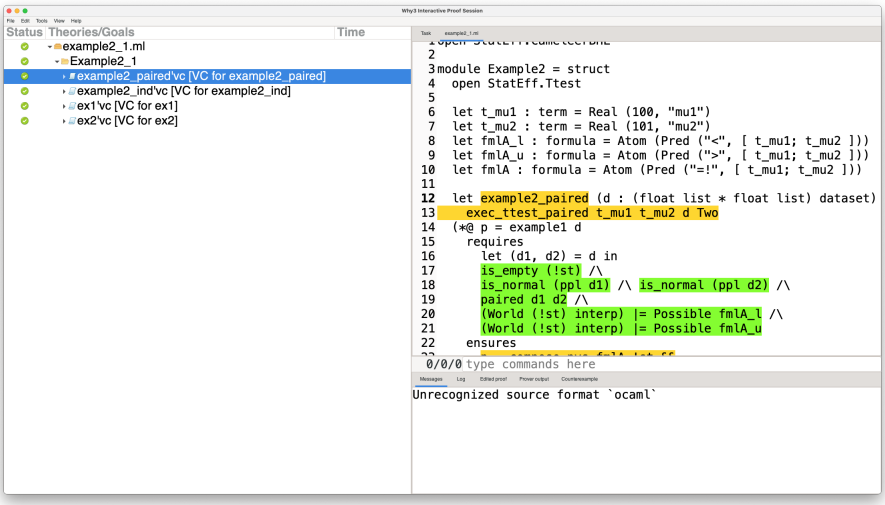
Non-paired *t*-test The code below shows the specification of the non-paired t-test:

```
val exec_ttest_ind_eq (r1 : term) (r2 : term) (y : dataset (list real,
    list real)) (alt : alternative) : real
writes { st }
requires {c'
  match y with
  | (y1, y2) ->
      is_normal (ppl y1) /\ is_normal (ppl y2) /\
      eq_variance (ppl y1) (ppl y2) /\ non_paired y1 y2 /\
```

```
      (exists r1':real_invisible. exists s:int. r1 = Real s r1') /\
      (exists r2':real_invisible. exists s:int. r2 = Real s r2') /\
      match alt with
      | Two ->
          (World !st interp) |= Possible (Atom (Pred "<" (Cons r1 (Cons
              r2 Nil)))) /\
          (World !st interp) |= Possible (Atom (Pred ">" (Cons r1 (Cons
              r2 Nil))))
      | Up ->
          (World !st interp) |= Not (Possible (Atom (Pred "<" (Cons r1 (
              Cons r2 Nil))))) /\
          (World !st interp) |= Possible (Atom (Pred ">" (Cons r1 (Cons
              r2 Nil))))
      | Low ->
          (World !st interp) |= Possible (Atom (Pred "<" (Cons r1 (Cons
              r2 Nil)))) /\
          (World !st interp) |= Not (Possible (Atom (Pred ">" (Cons r1 (
              Cons r2 Nil)))))
      end
end
}
ensures {
  let pv = pvalue () in
  result = pv /\
  let h = match alt with
    | Two -> Pred "=!" (Cons r1 (Cons r2 Nil))
    | Up -> Pred ">" (Cons r1 (Cons r2 Nil))
    | Low -> Pred "<" (Cons r1 (Cons r2 Nil))
  end in !st = !(old st) ++ Cons ("ttest_ind", Atom h, pv) Nil
}
```

This specification is the same as `exec_ttest_paired` except for the precondition `paired y1 y2`, which indicates `y1` and `y2` must be sampled in pairs, and the precondition `eq_variance (ppl y1) (ppl y2)`, which we will explain later.

We have code examples in `platform/examples/example2_1.ml`. To verify this code, execute `cameleer example2_1.ml`, and it will open Why3 IDE.



To execute this code, do the following:

```
$ cd platform/examples
$ dune exec -- ./example2_1.exe
```

The output will be as follows:

```
$ dune exec -- ./example2_1.exe
p-value (paired) : 0.070246
p-value (independent) : 0.027582
```

Equal vs Unequal Variance This section focuses on the *equal variance* assumption in the non-paired t-test. In the non-paired t-test, the assumption of equal variance is crucial; we should use different t-tests according to this assumption. When we know or assume that two populations have equal variance, we usually use *Student's t-test*. On the other hand, we apply *Welch's t-test* if we cannot assume equal variance. StatWhy distinguishes the difference by the precondition `eq_variance`.

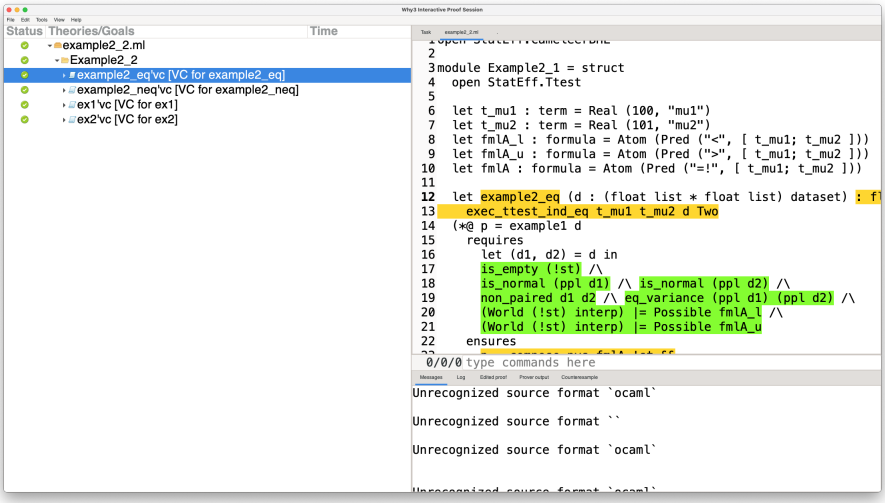
`exec_ttest_ind_eq` in the last section is our formalization of Student's t-test. The precondition `eq_variance (ppl y1) (ppl y2)` denotes that the two populations that respectively draw the datasets `y1` and `y2` have equal variance.

In contrast, `exec_ttest_ind_neq` below does not assume `eq_variance`:

```
val exec_ttest_ind_neq (r1 : term) (r2 : term) (y : dataset (list real,
    list real)) : real
  writes { st }
  requires {
    match y with
    | (y1, y2) ->
```

```
      is_normal (ppl y1) /\ is_normal (ppl y2) /\
      not (eq_variance (ppl y1) (ppl y2)) /\ non_paired y1 y2 /\
      (exists r1':real_invisible. exists s:int. r1 = Real s r1') /\
      (exists r2':real_invisible. exists s:int. r2 = Real s r2') /\
      (World !st interp) |= Possible (Atom (Pred "<" (Cons r1 (Cons r2
          Nil)))) /\
      (World !st interp) |= Possible (Atom (Pred ">" (Cons r1 (Cons r2
          Nil))))
  end
}
ensures {
  let pv = pvalue () in
  result = pv /\
  !st = !(old st) ++ Cons ("ttest_ind", Atom (Pred "=!" (Cons r1 (Cons
      r2 Nil))), pv) Nil
}
```

The code examples for these t-tests are available in `platform/examples/example2_2.ml`




```
$ cd platform/examples
$ dune exec -- ./example2_2.exe
p-value (equal variance) : 0.005385
p-value (unequal variance) : 0.022088
```

5.3 Example 3. Bonferroni correction in multiple comparison

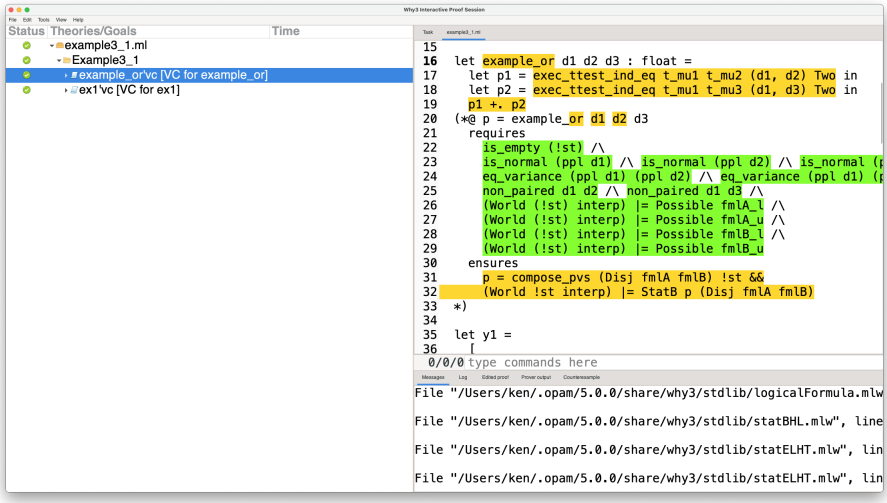
***p*-values of disjunctive alternative hypothesis** StatWhy also has the built-in Bonferroni correction. It can automatically check whether a program correctly calculates the p-value of a disjunctive alternative hypothesis using Bonferroni correction.

platform/examples/example3_1.ml

```
open StatEff.CameleerBHL

module Example1 = struct
  open StatEff.Ttest

  let t_mu1 : term = Real (100, "mu1")
  let t_mu2 : term = Real (101, "mu2")
  let t_mu3 : term = Real (102, "mu3")
  let fmlA_l : formula = Atom (Pred ("<", [ t_mu1; t_mu2 ]))
  let fmlA_u : formula = Atom (Pred (">", [ t_mu1; t_mu2 ]))
  let fmlA : formula = Atom (Pred ("=!", [ t_mu1; t_mu2 ]))
  let fmlB_l : formula = Atom (Pred ("<", [ t_mu1; t_mu3 ]))
  let fmlB_u : formula = Atom (Pred (">", [ t_mu1; t_mu3 ]))
  let fmlB : formula = Atom (Pred ("=!", [ t_mu1; t_mu3 ]))

  let example_or d1 d2 d3 : float =
    let p1 = exec_ttest_ind_eq t_mu1 t_mu2 (d1, d2) Two in
    let p2 = exec_ttest_ind_eq t_mu1 t_mu3 (d1, d3) Two in
    p1 +. p2
  (*@ p = example_or d1 d2 d3
    requires
      is_empty (!st) /\
      is_normal (ppl d1) /\ is_normal (ppl d2) /\ is_normal (ppl d3) /\
      eq_variance (ppl d1) (ppl d2) /\ eq_variance (ppl d1) (ppl d3) /\
      non_paired d1 d2 /\ non_paired d1 d3 /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u /\
      (World (!st) interp) |= Possible fmlB_l /\
      (World (!st) interp) |= Possible fmlB_u
    ensures
      p = compose_pvs (Disj fmlA fmlB) !st &&
      (World !st interp) |= StatB p (Disj fmlA fmlB)
  *)

  let y1 =
    [
      0.172199288696712305;
      1.56633241273914514;
      -1.46459260193754326;
      -1.43755987448188516;
      0.606597055392836;
      -0.858854622560882852;
      0.830966186381477567;
      -0.0902063463539077848;
      0.0151171772866927571;
      -1.29692214960254959;
      -1.49792102111270431;
      0.517798308025511522;
```

```
    -0.192132041057335529;
    -0.162761486272442774;
    -1.54871151183215727;
  ]

let y2 =
  [
    -0.21727268665710664;
    3.13557505149899818;
    0.771008947856724802;
    -0.334728073504754642;
    0.835158899203843;
    0.746551527960406;
    -0.436042372215016183;
    0.138553286012370269;
    0.681419038802263;
    1.33480596208851265;
    1.71624749449189817;
    1.19841148155714849;
    0.960676270885611583;
    1.63170098066680236;
    0.685939236361666427;
  ]

let y3 =
  [
    0.694010284447472481;
    0.466546449082268166;
    1.5077793567208897;
    0.416999624643256128;
    0.691373911805812202;
    -0.254342831838054839;
    0.512231009927806791;
    -0.405285516811907276;
    -0.160498199558324339;
    -1.32854303955898279;
    -0.26068133788479797;
    1.6438316627993137;
    0.174833054056991122;
    -0.172417656603847558;
    2.25569430033624485;
  ]

let ex1 () = example_or y1 y2 y3
(*@ requires
    is_empty !st /\
    is_normal (ppl y1) /\ is_normal (ppl y2) /\ is_normal (ppl y3) /\
    eq_variance (ppl y1) (ppl y2) /\ eq_variance (ppl y1) (ppl y3) /\
    non_paired y1 y2 /\ non_paired y1 y3 /\
    (World (!st) interp) |= Possible fmlA_1 /\
```

```
      (World (!st) interp) |= Possible fmlA_u /\
      (World (!st) interp) |= Possible fmlB_l /\
      (World (!st) interp) |= Possible fmlB_u *)

  let[@run] res =
    let res1, _ = run ex1 in
    Format.printf "p-value (or) : %f\n" res1
end
```




```
$ cd platform/examples/
$ dune exec -- ./example3_1.exe
p-value (or) : 0.052456
p-value (and) : 0.002064
```

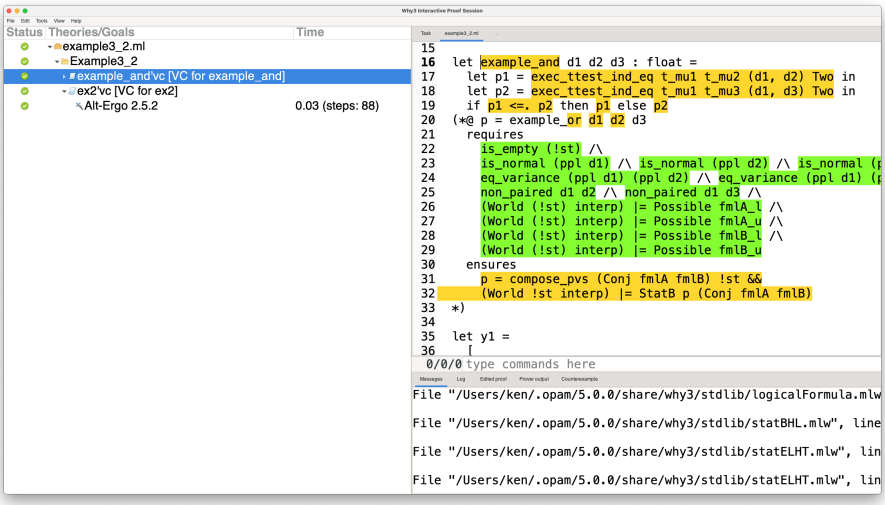
***p*-values of conjunctive hypotheses** For the calculation of the p-value of a conjunctive hypothesis (e.g., $\varphi \wedge \psi$), we take the minimum of the p-values of its subformulas (φ and ψ).

platform/examples/example3_2.ml

```
open StatEff.CameleerBHL

module Example2 = struct
  open StatEff.Ttest

  let t_mu1 : term = Real (100, "mu1")
  let t_mu2 : term = Real (101, "mu2")
  let t_mu3 : term = Real (102, "mu3")
```

```
let fmlA_l : formula = Atom (Pred ("<", [ t_mu1; t_mu2 ]))
let fmlA_u : formula = Atom (Pred (">", [ t_mu1; t_mu2 ]))
let fmlA : formula = Atom (Pred ("=!", [ t_mu1; t_mu2 ]))
let fmlB_l : formula = Atom (Pred ("<", [ t_mu1; t_mu3 ]))
let fmlB_u : formula = Atom (Pred (">", [ t_mu1; t_mu3 ]))
let fmlB : formula = Atom (Pred ("=!", [ t_mu1; t_mu3 ]))

let example_and d1 d2 d3 : float =
  let p1 = exec_ttest_ind_eq t_mu1 t_mu2 (d1, d2) Two in
  let p2 = exec_ttest_ind_eq t_mu1 t_mu3 (d1, d3) Two in
  if p1 <=. p2 then p1 else p2
(*@ p = example_or d1 d2 d3
  requires
    is_empty (!st) /\
    is_normal (ppl d1) /\ is_normal (ppl d2) /\ is_normal (ppl d3) /\
    eq_variance (ppl d1) (ppl d2) /\ eq_variance (ppl d1) (ppl d3) /\
    non_paired d1 d2 /\ non_paired d1 d3 /\
    (World (!st) interp) |= Possible fmlA_l /\
    (World (!st) interp) |= Possible fmlA_u /\
    (World (!st) interp) |= Possible fmlB_l /\
    (World (!st) interp) |= Possible fmlB_u
  ensures
    p = compose_pvs (Conj fmlA fmlB) !st &&
    (World !st interp) |= StatB p (Conj fmlA fmlB)
*)

let y1 =
  [
    0.172199288696712305;
    1.56633241273914514;
    -1.46459260193754326;
    -1.43755987448188516;
    0.606597055392836;
    -0.858854622560882852;
    0.830966186381477567;
    -0.0902063463539077848;
    0.0151171772866927571;
    -1.29692214960254959;
    -1.49792102111270431;
    0.517798308025511522;
    -0.192132041057335529;
    -0.162761486272442774;
    -1.54871151183215727;
  ]

let y2 =
  [
    -0.21727268665710664;
    3.13557505149899818;
    0.771008947856724802;
```

```
      -0.334728073504754642;
      0.835158899203843;
      0.746551527960406;
      -0.436042372215016183;
      0.138553286012370269;
      0.681419038802263;
      1.33480596208851265;
      1.71624749449189817;
      1.19841148155714849;
      0.960676270885611583;
      1.63170098066680236;
      0.685939236361666427;
    ]

  let y3 =
    [
      0.694010284447472481;
      0.466546449082268166;
      1.5077793567208897;
      0.416999624643256128;
      0.691373911805812202;
      -0.254342831838054839;
      0.512231009927806791;
      -0.405285516811907276;
      -0.160498199558324339;
      -1.32854303955898279;
      -0.26068133788479797;
      1.6438316627993137;
      0.174833054056991122;
      -0.172417656603847558;
      2.25569430033624485;
    ]

  let ex2 () = example_and y1 y2 y3
  (*@ requires
      is_empty !st /\
      is_normal (ppl y1) /\ is_normal (ppl y2) /\ is_normal (ppl y3) /\
      eq_variance (ppl y1) (ppl y2) /\ eq_variance (ppl y1) (ppl y3) /\
      non_paired y1 y2 /\ non_paired y1 y3 /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u /\
      (World (!st) interp) |= Possible fmlB_l /\
      (World (!st) interp) |= Possible fmlB_u *)

  let[@run] res =
    let res2, _ = run ex2 in
    Format.printf "p-value (and) : %f\n" res2
end
```

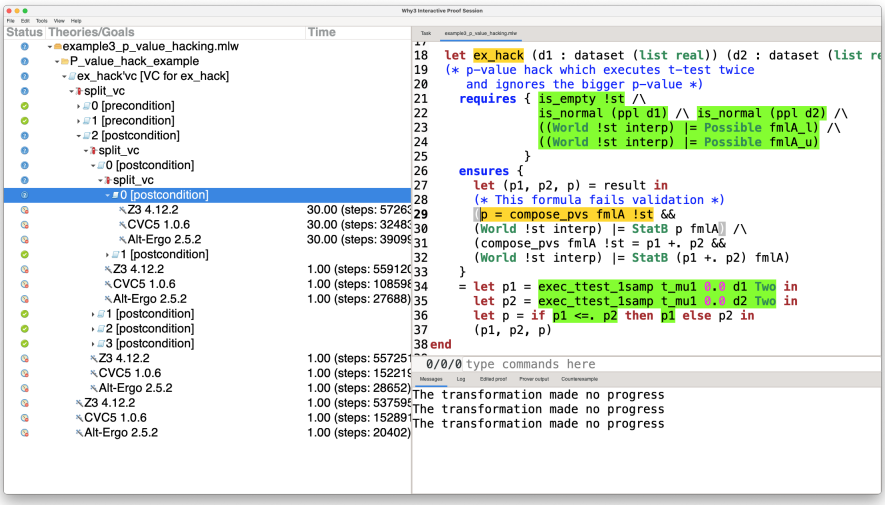


***p*-value hacking** In this example, we see that StatWhy does not validate the codes that report lower p-values than the actual ones (p-value hacking).

doc/example3_p_value_hacking.mlw

```
module P_value_hack_example

  val function mu1 : real_invisible

  let function t_mu1 : term = Real 104 mu1
  let function t_mu2 : term = RealConst 0.0

  let function afmlA_l : atomic_formula = (Pred "<" (Cons t_mu1 (Cons
      t_mu2 Nil)))
  let function afmlA_u : atomic_formula = (Pred ">" (Cons t_mu1 (Cons
      t_mu2 Nil)))
  let function afmlA : atomic_formula = (Pred "=!" (Cons t_mu1 (Cons
      t_mu2 Nil)))

  let function fmlA_l = Atom afmlA_l
  let function fmlA_u = Atom afmlA_u
  let function fmlA = Atom afmlA

  let ex_hack (d1 : dataset (list real)) (d2 : dataset (list real))
  (* p-value hack which executes t-test twice and ignores the bigger p-
      value *)
    requires { is_empty !st /\
               is_normal (ppl d1) /\ is_normal (ppl d2) /\
               ((World !st interp) |= Possible fmlA_l) /\
               ((World !st interp) |= Possible fmlA_u)
             }
```

```
    ensures {
      let (p1, p2, p) = result in
      (p = compose_pvs fmlA !st && (* This formula fails validation *)
      (World !st interp) |= StatB p fmlA) /\
      (compose_pvs fmlA !st = p1 +. p2 &&
      (World !st interp) |= StatB (p1 +. p2) fmlA)
    }
    = let p1 = exec_ttest_1samp t_mu1 0.0 d1 Two in
      let p2 = exec_ttest_1samp t_mu1 0.0 d2 Two in
      let p = if p1 <=. p2 then p1 else p2 in
      (p1, p2, p)
end
```



5.4 Example 4. Hypothesis tests for multiple group comparison

This example illustrates multiple comparisons, the methods that examine more than two data groups against each other. When conducting multiple comparisons, there is an increased chance of falsely claiming a statistically significant difference — false positives. To avoid this issue, we use several statistical methods to adjust for the number of comparisons performed.

For proper handling of multiple comparisons, StatWhy provides the specification of the one-way ANOVA (analysis of variance) to test for overall differences among the groups. It also includes the non-parametric Kruskal-Wallis test and the Alexander-Govern test as alternatives when the assumptions of ANOVA are not satisfied.

ANOVA We show the specification of the one-way ANOVA in StatWhy:

```
val exec_oneway (terms : list term) (ys : dataset (list (list real))) :
    real
  writes { st }
  requires {
    length terms = length ys /\
    length terms >= 2 /\ length ys >= 2 /\
    (forall t : term. mem t terms -> exists r:real_invisible, s:int. t =
        Real s r) /\
    (forall y : list real. mem y ys -> is_normal (ppl y)) /\
    (forall x y : dataset (list real). mem x ys /\ mem y ys ->
        eq_variance (ppl x) (ppl y)) /\
    (forall s t : term. mem s terms /\ mem t terms /\ not eq_term s t ->
      ((World !st interp) |= Possible (Atom (Pred "<" (Cons s (Cons t Nil
          )))) /\
      (World !st interp) |= Possible (Atom (Pred ">" (Cons s (Cons t Nil)
          )))))
  } ensures {
    let pv = pvalue () in
    result = pv /\
    let h = oneway_hypothesis terms in
    !st = !(old st) ++ Cons ("oneway", h, pv) Nil
  }
```

It is worth noting that the input arguments for terms and datasets used in the one-way ANOVA are lists, which allows for the comparisons of arbitrary finite groups of data.

The one-way ANOVA assumes the following conditions:

1. Each population of the samples is normally distributed.
2. All the populations have the same variance.

The former assumption is formalized as:

```
forall y. mem y ys -> is_normal (ppl y)
```

The latter is represented by:

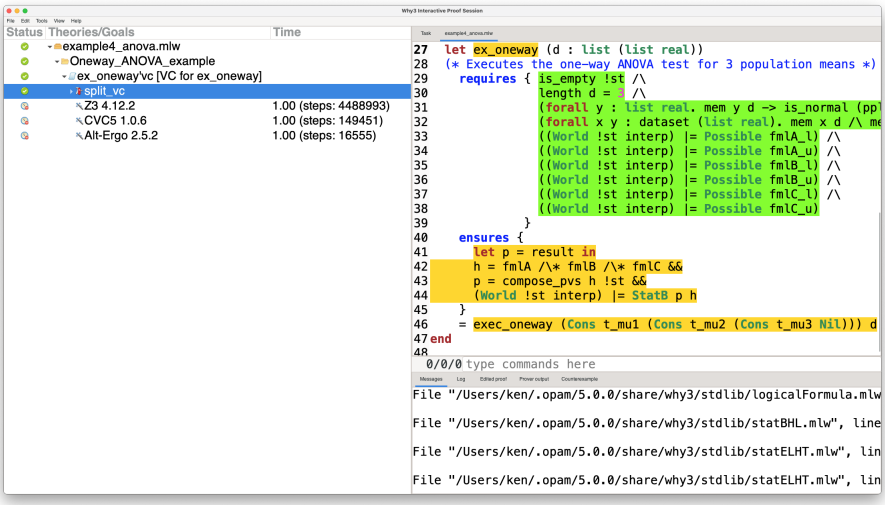
```
forall x y. mem x ys /\ mem y ys -> eq_variance (ppl x) (ppl y).
```

For the sake of brevity, we introduce an abbreviation that is not included in the WhyML syntax. In the above specification, the alternative hypothesis is abbreviated as `oneway_hypothesis terms`, which represents all the possible combinations of comparisons. For example, `oneway_hypothesis [termA; termB; termC]` represents the formula `termA =! termB" /\ "termA =! termC" /\ "termB =! termC"`.

doc/example4_anova.mlw

```
module Oneway_ANOVA_example
```

```
  use cameleerBHL.CameleerBHL
  use oneway.Oneway

  val function mu1 : real_invisible
  val function mu2 : real_invisible
  val function mu3 : real_invisible

  let function t_mu1 : term = Real 104 mu1
  let function t_mu2 : term = Real 105 mu2
  let function t_mu3 : term = Real 106 mu3

  let function fmlA_l = Atom (Pred "<" (Cons t_mu1 (Cons t_mu2 Nil)))
  let function fmlA_u = Atom (Pred ">" (Cons t_mu1 (Cons t_mu2 Nil)))
  let function fmlA  = Atom (Pred "=!" (Cons t_mu1 (Cons t_mu2 Nil)))

  let function fmlB_l = Atom (Pred "<" (Cons t_mu1 (Cons t_mu3 Nil)))
  let function fmlB_u = Atom (Pred ">" (Cons t_mu1 (Cons t_mu3 Nil)))
  let function fmlB = Atom (Pred "=!" (Cons t_mu1 (Cons t_mu3 Nil)))

  let function fmlC_l = Atom (Pred "<" (Cons t_mu2 (Cons t_mu3 Nil)))
  let function fmlC_u = Atom (Pred ">" (Cons t_mu2 (Cons t_mu3 Nil)))
  let function fmlC = Atom (Pred "=!" (Cons t_mu2 (Cons t_mu3 Nil)))

  let function h = oneway_hypothesis (Cons t_mu1 (Cons t_mu2 (Cons t_mu3
      Nil)))

  let ex_oneway (d : list (list real))
  (* Executes the one-way ANOVA test for 3 population means *)
    requires { is_empty !st /\
               length d = 3 /\
               (forall y : list real. mem y d -> is_normal (ppl y)) /\
               (forall x y : dataset (list real). mem x d /\ mem y d ->
                   eq_variance (ppl x) (ppl y)) /\
               ((World !st interp) |= Possible fmlA_l) /\
               ((World !st interp) |= Possible fmlA_u) /\
               ((World !st interp) |= Possible fmlB_l) /\
               ((World !st interp) |= Possible fmlB_u) /\
               ((World !st interp) |= Possible fmlC_l) /\
               ((World !st interp) |= Possible fmlC_u)
             }
    ensures {
      let p = result in
      h = fmlA /\* fmlB /\* fmlC &&
      p = compose_pvs h !st &&
      (World !st interp) |= StatB p h
    }
    = exec_oneway (Cons t_mu1 (Cons t_mu2 (Cons t_mu3 Nil))) d
end
```

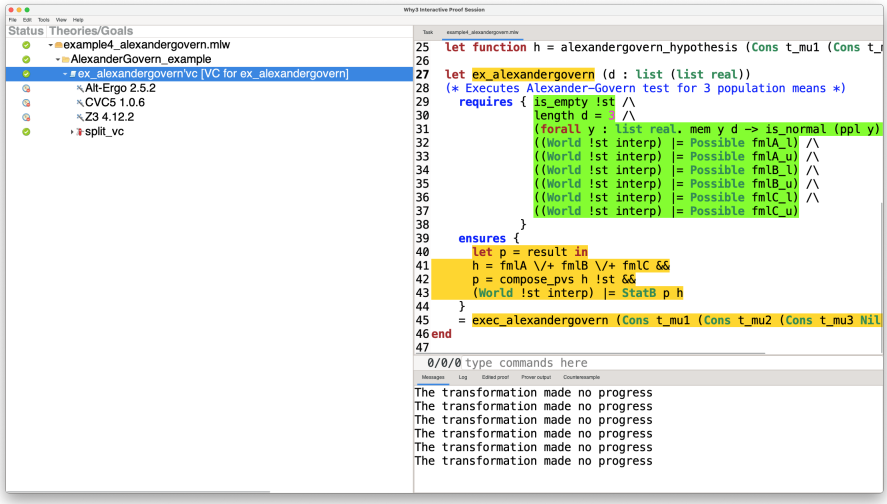


Alexander-Govern test The Alexander-Govern test requires that each sample comes from a normally distributed population but relaxes the assumption of equal variance on ANOVA. We show an example of the Alexander-Govern test as follows:

doc/example4_alexandergovern.mlw

```
module AlexanderGovern_example
  use cameleerBHL.CameleerBHL
  use alexandergovern.AlexanderGovern

  val function mu1 : real_invisible
  val function mu2 : real_invisible
  val function mu3 : real_invisible

  let function t_mu1 : term = Real 104 mu1
  let function t_mu2 : term = Real 105 mu2
  let function t_mu3 : term = Real 106 mu3

  let function fmlA_l = Atom (Pred "<" (Cons t_mu1 (Cons t_mu2 Nil)))
  let function fmlA_u = Atom (Pred ">" (Cons t_mu1 (Cons t_mu2 Nil)))
  let function fmlA  = Atom (Pred "=!" (Cons t_mu1 (Cons t_mu2 Nil)))

  let function fmlB_l = Atom (Pred "<" (Cons t_mu1 (Cons t_mu3 Nil)))
  let function fmlB_u = Atom (Pred ">" (Cons t_mu1 (Cons t_mu3 Nil)))
  let function fmlB = Atom (Pred "=!" (Cons t_mu1 (Cons t_mu3 Nil)))

  let function fmlC_l = Atom (Pred "<" (Cons t_mu2 (Cons t_mu3 Nil)))
  let function fmlC_u = Atom (Pred ">" (Cons t_mu2 (Cons t_mu3 Nil)))
  let function fmlC = Atom (Pred "=!" (Cons t_mu2 (Cons t_mu3 Nil)))
```

```
  let function h = alexandergovern_hypothesis (Cons t_mu1 (Cons t_mu2 (
      Cons t_mu3 Nil)))

  let ex_alexandergovern (d : list (list real))
  (* Executes Alexander-Govern test for 3 population means *)
    requires { is_empty !st /\
               length d = 3 /\
               (forall y : list real. mem y d -> is_normal (ppl y)) /\
               ((World !st interp) |= Possible fmlA_l) /\
               ((World !st interp) |= Possible fmlA_u) /\
               ((World !st interp) |= Possible fmlB_l) /\
               ((World !st interp) |= Possible fmlB_u) /\
               ((World !st interp) |= Possible fmlC_l) /\
               ((World !st interp) |= Possible fmlC_u)
             }
    ensures {
      let p = result in
      h = fmlA \/+ fmlB \/+ fmlC &&
      p = compose_pvs h !st &&
      (World !st interp) |= StatB p h
    }
    = exec_alexandergovern (Cons t_mu1 (Cons t_mu2 (Cons t_mu3 Nil))) d
end
```

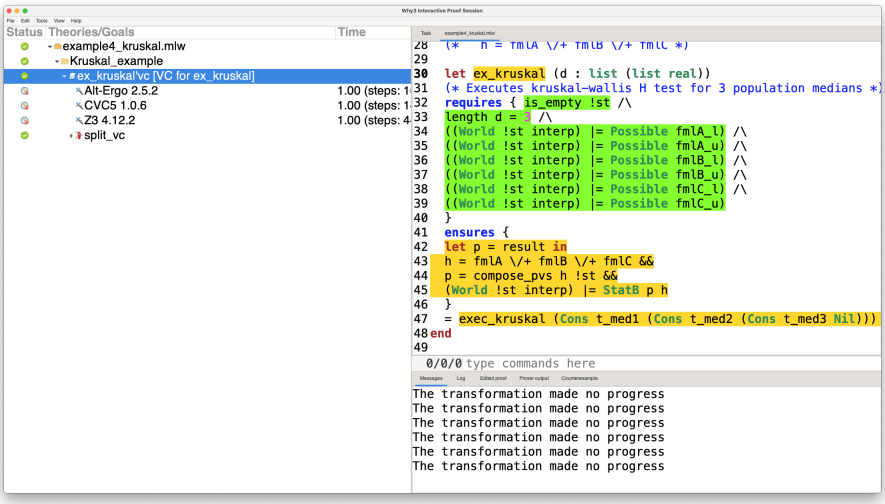


Kruskal-Wallis test The Kruskal-Wallis H-test is a non-parametric variant of ANOVA; it is not assumed that each sample is from normally distributed populations with equal variance. Here is an example of the Kruskal-Wallis H-test whose null hypothesis is that all medians of three populations are equal:

doc/example4_kruskal.mlw

```
module Kruskal_example
  use cameleerBHL.CameleerBHL
  use kruskal.Kruskal

  val function med1 : real_invisible
  val function med2 : real_invisible
  val function med3 : real_invisible

  let function t_med1 : term = Real 104 med1
  let function t_med2 : term = Real 105 med2
  let function t_med3 : term = Real 106 med3

  let function fmlA_l = Atom (Pred "<" (Cons t_med1 (Cons t_med2 Nil)))
  let function fmlA_u = Atom (Pred ">" (Cons t_med1 (Cons t_med2 Nil)))
  let function fmlA  = Atom (Pred "=!" (Cons t_med1 (Cons t_med2 Nil)))

  let function fmlB_l = Atom (Pred "<" (Cons t_med1 (Cons t_med3 Nil)))
  let function fmlB_u = Atom (Pred ">" (Cons t_med1 (Cons t_med3 Nil)))
  let function fmlB = Atom (Pred "=!" (Cons t_med1 (Cons t_med3 Nil)))

  let function fmlC_l = Atom (Pred "<" (Cons t_med2 (Cons t_med3 Nil)))
  let function fmlC_u = Atom (Pred ">" (Cons t_med2 (Cons t_med3 Nil)))
  let function fmlC = Atom (Pred "=!" (Cons t_med2 (Cons t_med3 Nil)))

  let function h = kruskal_hypothesis (Cons t_med1 (Cons t_med2 (Cons
      t_med3 Nil)))

  (* lemma lem_hypothesis: *)
  (*   h = fmlA \/+ fmlB \/+ fmlC *)

  let ex_kruskal (d : list (list real))
  (* Executes kruskal-wallis H test for 3 population medians *)
    requires { is_empty !st /\
               length d = 3 /\
               ((World !st interp) |= Possible fmlA_l) /\
               ((World !st interp) |= Possible fmlA_u) /\
               ((World !st interp) |= Possible fmlB_l) /\
               ((World !st interp) |= Possible fmlB_u) /\
               ((World !st interp) |= Possible fmlC_l) /\
               ((World !st interp) |= Possible fmlC_u)
             }
    ensures {
      let p = result in
      h = fmlA \/+ fmlB \/+ fmlC &&
      p = compose_pvs h !st &&
      (World !st interp) |= StatB p h
    }
    = exec_kruskal (Cons t_med1 (Cons t_med2 (Cons t_med3 Nil))) d
```

```
end
```



5.5 Experimental Results

We have verified the code examples in this section on a MacBook Pro with Apple M2 Max CPU and 96 GB memory using Alt-Ergo 2.5.2, CVC5 1.0.6, and Z3 4.12.2. For each verification condition, we set the limits for the execution time and the memory to 120 seconds and 10 GB, respectively. Table 1 summarizes the experimental results.

References

1. Charguéraud, A., Filliâtre, J., Lourenço, C., Pereira, M.: GOSPEL - providing ocaml with a formal specification language. In: Proc. of the 24th International Symposium on Formal Methods (FM 2019). Lecture Notes in Computer Science, vol. 11800, pp. 484–501. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_29
2. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Proc. of the 22nd European Symposium on Programming (ESOP 2013). Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
3. Kawamoto, Y., Sato, T., Suenaga, K.: Formalizing statistical beliefs in hypothesis testing using program logic. In: Proc. KR'21. pp. 411–421 (2021). https://doi.org/10.24963/kr.2021/39
4. Kawamoto, Y., Sato, T., Suenaga, K.: Sound and relatively complete belief hoare logic for statistical hypothesis testing programs. Artif. Intell. **326**, 104045 (2024). https://doi.org/10.1016/J.ARTINT.2023.104045

Table 1: The experimental results of code examples. The time (in seconds) it took to verify with each solver is displayed for each VC. A blank shows that the solver did not provide a conclusive answer. "> 120" indicates a timeout. For three codes in Example 4., the total time taken to verify the partial VCs with a solver is used as the overall verification time because these codes can be proved only by "auto level 3."

File	VC	CVC5	Alt-Ergo	Z3
example1.ml	example1'vc	0.14	0.39	
	ex'vc	0.03	0.02	0.01
example2_1.ml	example2_paired'vc	0.14	1.06	
	example2_ind'vc	0.12	1.01	
	ex1'vc	0.03	0.02	0.02
	ex2'vc	0.03	0.03	0.02
example2_2.ml	example2_eq'vc	0.13	1.01	
	example2_neq'vc	0.12	1.01	
	ex1'vc	0.03	0.03	0.02
	ex2'vc	0.04	0.04	0.02
example3_1.ml	example_or'vc	1.79	>120	
	ex1'vc	0.03	0.03	0.02
example3_2.ml	example_and'vc	1.64	>120	
	ex2'vc	0.04	0.03	0.02
example4_anova.mlw	ex_oneway'vc	4.81	>120	>120
example4_kruskal.mlw	ex_kruskal'vc	4.41	>120	>120
example4_alexandergovern.mlw	ex_alexandergovern'vc	4.54	>120	>120

Table 2: The experimental results of examples.

		VC1			**VC2**		
File	Hypothesis test	CVC5	Alt-Ergo	Z3	CVC5	Alt-Ergo	Z3
example1.ml	One-sample t-test	0.14	0.39		0.03	0.02	0.01
example2_1.ml	Paired t-test	0.14	1.06		0.03	0.02	0.02
	Student's t-test	0.12	1.01		0.03	0.03	0.02
example2_2.ml	Student's t-test	0.13	1.01		0.03	0.03	0.02
	Welch's t-test	0.12	1.01		0.04	0.04	0.02
example3_1.ml	Disjunctive combination	1.79	>120		0.03	0.03	0.02
example3_2.ml	Conjunctive combination	1.64	>120		0.04	0.03	0.02
example4_anova.mlw	ANOVA	4.81	>120	>120			
example4_kruskal.mlw	Kruskal-Wallis test	4.41	>120	>120			
example4_alexandergovern.mlw	Alexander-Govern test	4.54	>120	>120			

Table 3: The experimental results of hypothesis testing commands.

		VC1			**VC2**	
Hypothesis test	Lines of code	CVC5	Alt-Ergo	Z3	CVC5	Alt-Ergo	Z3
One-sample t-test	12	0.14	0.39		0.03	0.02	0.01
Paired t-test	14	0.14	1.06		0.03	0.02	0.02
Student's t-test	14	0.12	1.01		0.03	0.03	0.02
Welch's t-test	14	0.12	1.01		0.04	0.04	0.02
ANOVA	20	4.81	>120	>120			
Kruskal-Wallis test	18	4.41	>120	>120			
Alexander-Govern test	19	4.54	>120	>120			

Table 4: The table of the verification time versus the number of hypotheses that comprise a disjunctive or conjunctive hypothesis.

Number of hypotheses	OR	AND
1	0.11	0.11
2	1.56	1.21
3	17.83	11.05
4	>120	>120

5. Pereira, M., Ravara, A.: Cameleer: A deductive verification tool for ocaml. In: Proc. of the 33rd International Conference on Computer Aided Verification (CAV 2021), Part II. Lecture Notes in Computer Science, vol. 12760, pp. 677–689. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_31