User Documentation for StatWhy v.1.3.0

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

[1] National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan
[2] University of Tsukuba, Ibaraki, Japan
[3] Kyoto University, Kyoto, Japan

Abstract. StatWhy is a software tool for automatically verifying the correctness of statistical hypothesis testing programs. Specifically, programmers are required to annotate the source code of the statistical programs with the requirements for the statistical analyses. Then our StatWhy tool automatically checks whether the programmers have properly specified the requirements for the statistical methods, thereby identifying any missing or incorrect requirements that need to be corrected. In this documentation, we first present how to install and use the StatWhy tool. We then demonstrate how the tool can avoid common errors in the application of a variety of hypothesis testing methods.

1 Foreword

Statistical methods have been widely misused and misinterpreted in various scientific fields, raising significant concerns about the integrity of scientific research. To mitigate this problem, we have proposed a new method for formally specifying and automatically verifying the correctness of statistical programs in our paper [3].

StatWhy is a software tool that implements our proposed method for automatically checking whether a programmer has properly specified the requirements for the statistical methods in a source code.

In Section 2, we first present how to install the StatWhy tool. In Section 3, we present how to execute StatWhy. In Section 4, we show an illustrating example to see how a tool user can verify a statistical program. In Section 5, we briefly explain notations used in StatWhy specifications. In Section 6, we present examples to demonstrate how we can prevent common errors in applying a variety of hypothesis testing methods in programs In Section 7, we show the list of all the hypothesis testing methods supported in StatWhy and provide an experimental evaluation [3] of the performance of the program verification using StatWhy.

1.1 Availability

The source code of the StatWhy tool is publicly available at https://github.com/fm4stats/statwhy including a range of example programs.

⋆ The authors are listed in alphabetical order.

1.2 Contact

Report any bugs and requests to https://github.com/fm4stats/statwhy/issues.

1.3 Updates from 1.2.0

– We renamed the type `population` to `distribution` to use terminology correctly.
– We added a `scale` parameter to the type `dataset`. This allows StatWhy to check that a measurement scale is correctly specified when a hypothesis test is used.
– We implemented more hypothesis testing methods. See Section **??** for the list of supported methods.
– We refined our custom proof strategy "StatWhy". The original proof strategy is available as "StatWhy aggressive".

1.4 Acknowledgments.

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

2 Installation

In this section, we explain how to install StatWhy and our extension of Cameleer [6].

2.1 Installing StatWhy

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

1. Install opam
   On Ubuntu:

```
$ sudo apt-get update
$ sudo apt-get install opam
```

   On macOS, 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
   To install OCaml 5.0, execute the following command:[4]

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

Installing StatWhy Download the source code of StatWhy, including an extension of Cameleer. Install StatWhy by running:

```
$ unzip statwhy-1.3.0.zip
$ cd statwhy-1.3.0/cameleer
$ opam pin add .
```

This will install StatWhy (included in "cameleer/src/statwhy") and their dependencies, including Cameleer. In the installation of Cameleer, the Why3 platform [2] is automatically installed.[5]

Installing cvc5 On Ubuntu 24.04 LTS, you can install cvc5 by:

```
$ sudo apt install cvc5
```

Alternatively, you can directly download a binary from the GitHub repository:

```
$ wget https://github.com/cvc5/cvc5/releases/download/cvc5-1.2.0/cvc5-Linux-
    x86_64-static.zip
$ unzip cvc5-Linux-x86_64-static.zip
$ sudo cp ./cvc5-Linux-x86_64-static/bin/cvc5 /usr/local/bin
```

On macOS, a Homebrew Tap for cvc5 is also available:

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

After installing the solver, run the following command to let Why3 detect it:

```
$ why3 config detect
```

Remarks on executing hypothesis testing programs in OCaml To execute `.ml` files in examples/executable_examples, you need to install the following additional packages:

1. pyml (OCaml package)

[4] Using the versions later than OCaml 5.0, we encountered a dynamic link issue with Cameleer on macOS.

[5] We have tested StatWhy using Why3 1.8.0 and Cameleer 0.1.

2. scipy (Python library)

To install these packages, execute the following commands:

```
$ opam install pyml
$ pip install scipy
```

Remarks on reinstalling StatWhy If you update from an older version of StatWhy or have updated CVC5 since your last StatWhy installation, we recommend deleting the old /.statwhy.conf file. Upon the next launch of StatWhy, a new config file will be generated.

3 Usage

You can verify an OCaml code via Cameleer by running the following:

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

If you want to verify a WhyML code:

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

Note that in both cases, our extension of Cameleer is required to load StatWhy.

```
statwhy <file>.(ml|mlw)
Verify (OCaml|WhyML) program

  -L add <dir> to the search path
  --genconf generate config file
  --debug print debug information
  --batch activate batch mode
  --extract activate extraction mode
  --prover set prover for batch mode
  --version  print version information
```

4 Getting Started

We show an example of an OCaml program annotated with preconditions and postconditions that we want to verify using StatWhy. 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.

examples/example1.ml

```
open CameleerBHL

module Example1 = struct
  open Ttest
  (* Declarations of a distribution and formulas *)
  let t_n = NormalD (Param "mu1", Param "var")
  let fmlA_l = mean t_n $< const_term 1.0
  let fmlA_u = mean t_n $> const_term 1.0
  let fmlA = mean t_n $!= const_term 1.0

  (* executes the t-test for a population mean *)
  let example1 (d : float dataset) : float = exec_ttest_1samp t_n 1.0 d Two
  (*@ p = example1 d
    requires
      is_empty (!st) /\
      sampled d t_n /\
      d.scale = Interval /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u
    ensures
      Eq p = compose_pvs fmlA !st &&
      (World !st interp) |= StatB (Eq p) fmlA
  *)

  ...
end
```

The specification of `example1` is written in the Gospel specification language [1]. The comment section starting with `(*@` denotes the specification of `example1`.[6] More information on the syntax of Gospel can be found on the following webpage: https://ocaml-gospel.github.io/gospel/language/syntax.

Next, we explain the details of the function `example1` as follows.

```
  let example1 (d : float dataset) : float = exec_ttest_1samp t_n 1.0 d Two
  (*@ p = example1 d
    requires
      is_empty (!st) /\
      sampled d t_n /\
      d.scale = Interval /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u
    ensures
      Eq p = compose_pvs fmlA !st &&
      (World !st interp) |= StatB (Eq p) fmlA
  *)
```

In this program, given a dataset `d` as input, the command `exec_ttest_1samp t_n 1.0 d Two` computes the p-value of the one-sample t-test with the alternative hypothesis `fmlA` that the mean of the population distribution `t_n` (from which the dataset `d` was sampled) is not 1.0.

At the beginning of the specification, `p = example1 d` assigns the result of `example1 d` to the name `p`, which can be used throughout the specification.

The `requires` clause describes the *precondition* for correctly applying the t-test.

[6] Programmers are required to use the WhyML language to describe specifications.

– `is_empty (!st)` specifies that the record `st` of hypothesis tests is empty. This requirement reminds the programmer to check that no hypothesis test has been conducted before this test.
– `sampled d t_n` means that the dataset `d` has been sampled from a normal distribution `t_n`. The values of `t_n`'s parameters (mean `mu1` and variance `var`) are not specified in the specification. This condition prevents the programmer from forgetting to check whether the population follows a normal distribution.
– `d.scale = Interval`[7] specifies that the interval scale is used in this t-test on the dataset `d`. This condition ensures that a dataset with the correct scale of measure is applied to the hypothesis test.
– `(World (!st) interp) |= Possible fmlA_l` and `(World (!st) interp) |= Possible fmlA_u` represent that the analyst has a prior belief that the *lower-tail* hypothesis `fmlA_l` $\overset{\text{def}}{=}$ $(\mathsf{mean}(x) < 1.0)$ may be true, and that the *upper-tail* hypothesis `fmlA_u` $\overset{\text{def}}{=}$ $(\mathsf{mean}(x) > 1.0)$ may be true[8]. Thanks to this annotation, programmers are reminded to check whether the alternative hypothesis should be two-tailed or one-tailed, and thus whether the t-test command should be two-tailed or one-tailed.

The `ensures` clause describes the *postcondition*, i.e., what the analyst wants to learn from the t-test in the world where the test has been performed.

– `(Eq p) = compose_pvs fmlA !st` represents that `p` is equal to the p-value obtained by this hypothesis test with the alternative hypothesis `fmlA`.
– `(World !st interp) |= StatB p fmlA` represents that the analyst obtains a statistical belief on the alternative hypothesis `fmlA` with the p-value `p`, in the world equipped with the record `st` of all hypothesis tests executed so far. This logical formula (`StatB p fmlA`) employs a statistical belief modality `StatB` introduced in belief Hoare logic (BHL) [4,5].
– The logical connective `&&` represents an *asymmetric conjunction* to control the goal-splitting transformation; the proof task for A `&&` B is split into those for A and $A \to B$.

For the instructions on verifying this code, see Section 6.1.

Remarks on Modules and Expressions

We present remarks on OCaml programs to be verified using StatWhy.

– We need to open `CameleerBHL` and `Ttest` (or any hypothesis testing modules) explicitly.

[7] `d.scale` can denote either `Nominal`, `Ordinal`, `Interval`, `Rational`, or `Unspecified`. To perform a hypothesis test, we need to specify a scale appropriate to the hypothesis testing method and the situation. We set `d.scale` as `Unspecified` if the dataset `d` represents a histogram or a contingency table.

[8] We remark that the disjunction `fmlA_l` $\vee$ `fmlA_u` is logically equivalent to `fmlA`.

– We add an 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.
– We cannot use the "and" pattern `_` and the "unit" pattern `()` simultaneously in top-level definitions. The current version of Cameleer does not support these patterns in top-level definitions. For instance, we cannot use the expression of the form "`let[@run] _ = ...`" in a program.

5 Notations in StatWhy Specifications

In this section, we briefly describe notations used in StatWhy specifications. As shown in the previous section, we use formulas, such as `(World (!st) interp) |= Possible fmlA_l`, to specify the requirements of a hypothesis testing program in the framework of Belief Hoare logic (BHL) [4,5]. These formulas represent certain properties of populations or datasets that express the preconditions and postconditions for correctly applying hypothesis tests in programs.

5.1 Overview of belief Hoare Logic (BHL)

Belief Hoare logic (*BHL*) [4,5] is a program logic equipped with epistemic modal operators for the statistical beliefs acquired via hypothesis testing. We briefly explain this logic using a simple example described in our paper [3] as follows.

In the framework of BHL, we express a procedure for statistical hypothesis testing as a program C using a programming language. Then, we use modal logic to describe the requirements for the tests as a *precondition* formula, e.g.,

$$\psi_{\mathsf{pre}} \stackrel{\text{def}}{=} y \leftsquigarrow N(\mu, \sigma^2) \wedge \mathbf{P}\varphi \wedge \kappa_{\emptyset},$$

where $y \leftsquigarrow N(\mu, \sigma^2)$ represents that a dataset y is sampled from the population that follows a normal distribution $N(\mu, \sigma^2)$ with an unknown mean μ and an unknown variance σ^2. The modal formula $\mathbf{P}\varphi$ represents that, before conducting the hypothesis test, we have the *prior belief* that the alternative hypothesis φ *may be* true. The formula $\kappa_{\emptyset}$ represents that no statistical hypothesis testing has been conducted previously.

The statistical belief we acquire from the hypothesis test is specified as a *postcondition* formula, e.g.,

$$\psi^{\mathsf{post}} \stackrel{\text{def}}{=} \mathbf{K}^{\leq 0.05}_{y,A}\varphi. \tag{1}$$

Intuitively, by a hypothesis test A on the dataset y, we believe φ with a p-value $\alpha \leq 0.05$. Since the result of the hypothesis test may be wrong, we use the belief modality $\mathbf{K}^{\leq 0.05}_{y,A}$ instead of the S5 knowledge modality $\mathbf{K}$. Using this logic, the interpretation of the result of statistical methods is regarded to be epistemic.

Finally, we combine all the above and describe the whole statistical inference as a *judgment*:

$$\mathit{\Gamma} \vdash \{\psi_{\mathsf{pre}}\}\ C\ \{\psi^{\mathsf{post}}\}, \tag{2}$$

representing that whenever the precondition ψ_{pre} is satisfied, the execution of the program C results in the satisfaction of the postcondition ψ^{post}. By deriving this judgment using derivation rules in BHL, we conclude that the procedure for the statistical inference is correct whenever the precondition is satisfied.

5.2 BHL Formulas

To describe the requirements and interpretations of statistical analyses in Gospel, we introduce types for *terms*, *atomic formulas*, and *logical formulas* of an extension of belief Hoare logic (BHL) as follows: [9]

```
type term = RealT real_term | DistributionT distribution | ...
type atomic_formula = Pred psymb (list term)
type formula = Atom atomic_formula | Not formula
             | Conj formula formula | Disj formula formula
             | Possible formula | Know formula | StatB pvalue formula
             | ...
```

where a term can express a real number and a distribution; an atomic formula consists of a predicate symbol and a list of terms; a BHL formula is built using modal epistemic operators `Possible`, `Know`, and `StatB`. Then, we introduce predicate symbols (e.g., `eq_variance` and `check_variance`), functions symbols (e.g., `mean` and `ppl`), and a Kripke semantics where BHL formulas are interpreted under (i) the record `st` of all hypothesis tests executed so far [4,5] and (ii) the interpretation `interp` of private variables.

The interpretation of a BHL formula, `fml`, in a possible world, `World !st interp`, is written as `(World !st interp) |= fml`. The two symbols `st` and `interp` are reserved words defined in `cameleerBHL.mlw`.

Predicate and function symbols are defined in the files `logicalFormula.mlw` and `atomicFormulas.mlw`. BHL and the Kripke semantics are implemented in `statBHL.mlw` and `statELHT.mlw`.

5.3 Useful Operators to Describe Hypothesis Testing Programs

We introduce useful operators to facilitate describing the specification of StatWhy programs. Since hypothesis testing programs often involve comparisons among multiple groups of data, the preconditions and postconditions can become lengthy by repeating similar conditions. To simplify such redundant specifications, we have provided a set of folding operations, allowing programmers to abstract away repetitive parts.[10]

For example, to simplify the conditions for comparing each pair of groups, we can use the operator `for_all`. This higher-order function checks whether a given predicate holds for all elements in a list. In StatWhy programs, this operator is often used to describe assertions that must hold for combinations of distributions or terms. In the following example, the formula asserts that all the distributions have the same variance:

[9] The actual definition can be found in `logicalFormula.mlw`.
[10] These folding operations are defined in `hof.mlw` and `utility.mlw`.

```
for_all
(fun t -> let (x, y) = t in
  (World !st interp) |= eq_variance x y)
(cmb dists)
```

In the above formula, `for_all` is used to describe the iteration over all possible pairs `(x, y)` of distributions in a set `dists`. Using the predicate `eq_variance`, this formula states that for each pair `(x, y)` of the distributions, `x` and `y` have the same variance.

To improve the performance of StatWhy, we implemented a custom proof strategy—a combination of proof tactics and transformations—to accelerate the proof search in the presence of folding operations. Specifically, by applying a transformation called `compute_specified`, StatWhy automatically *unfolds* assertions written with syntax sugar such as `for_all` before discharging a verification condition. For instance, the above example formula is first transformed into:

```
((World !st interp) |= eq_variance p_1 p_2)
/\ ((World !st interp) |= eq_variance p_1 p_3)
/\ ... /\ ((World !st interp) |= eq_variance p_(n-1) p_n)
```

Then, thanks to the removal of the folding operation, StatWhy can efficiently discharge the verification conditions even when the set `dists` of distributions is large.[11]

Programmers can also choose to use other predefined abbreviations available in `hof.mlw` and `utility.mlw`.

- `hof.mlw` provides typical higher-order operations on polymorphic lists, such as fold and map.
- `utility.mlw` file defines operations that are useful to handle combinations of terms and distributions, which are frequently used in multiple comparison programs.

For example, the `cmb` function in `utility.mlw` enables users to compute all possible pairings of elements in a polymorphic list. This function is particularly useful in StatWhy when multiple distributions or terms need to be combined to describe specifications. A concrete example of using `cmb` appeared in the previous example, where `(cmb dists)` represents the set of all possible pairs of the distributions contained in `dists`.

Moreover, some hypothesis tests, such as the one-way ANOVA and Dunnett's test, have useful functions for defining alternative hypotheses, as it might be too tedious to define manually. Thanks to our custom proof strategy, these functions are also unfolded before discharging the goals, ensuring the proof automation remains effective.

[11] Conventionally, the number of groups compared in a hypothesis test is usually less than 8. However, we have found that SMT solvers often get stuck when trying to discharge the verification conditions that involve such folding operations.

6 Examples of Analyses Using StatWhy

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

- Section 6.1 demonstrates how to use StatWhy in an example of a one-sample t-test.
- Section 6.2 shows how StatWhy checks whether appropriate variants of t-test are applied to different situations.
- Section 6.3 explains how StatWhy verifies a program for the Bonferroni correction in multiple comparisons and addresses p-value hacking problems.
- Section 6.4 shows how StatWhy checks the programs with ANOVA and other hypothesis tests under different requirements.
- Section 6.5 demonstrates how StatWhy verifies Tukey's HSD test—a method for multiple comparison.

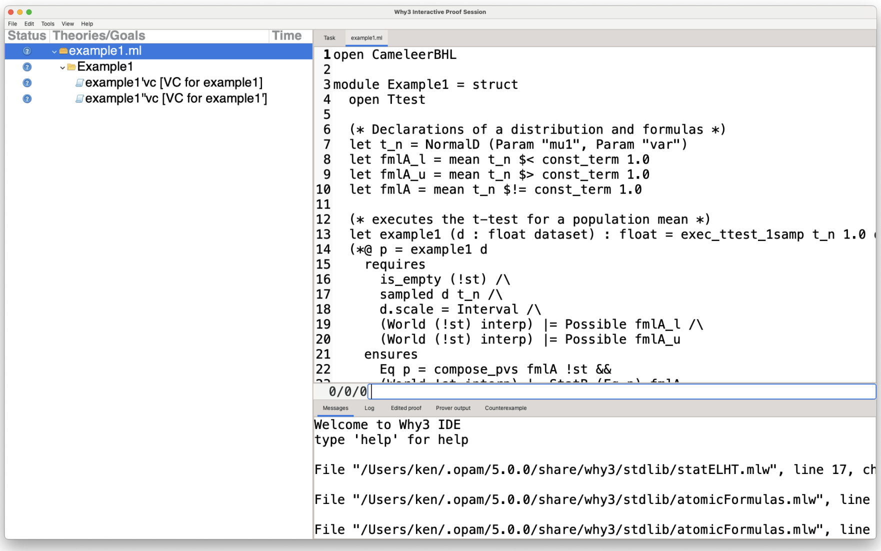
6.1 Simple t-test (One-Sample t-test)

We demonstrate how to use StatWhy through an example of verifying a program that performs the *t-test for a mean of a population*.

To verify the OCaml program `examples/example1.ml`, execute the following command:

```
$ statwhy examples/example1.ml
```

This command transforms the OCaml code into WhyML code, generates the verification conditions (VC), and launches the Why3 IDE as follows.



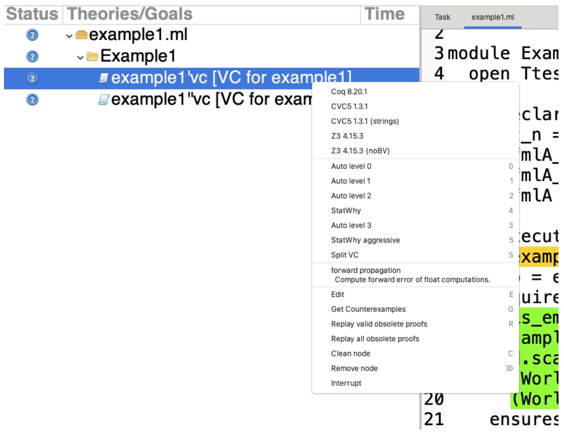
```
example1.ml
  Example1
    example1'vc [VC for example1]
    example1''vc [VC for example1']
```

```
open CameleerBHL

module Example1 = struct
  open Ttest

  (* Declarations of a distribution and formulas *)
  let t_n = NormalD (Param "mu1", Param "var")
  let fmlA_l = mean t_n $< const_term 1.0
  let fmlA_u = mean t_n $> const_term 1.0
  let fmlA = mean t_n $!= const_term 1.0

  (* executes the t-test for a population mean *)
  let example1 (d : float dataset) : float = exec_ttest_1samp t_n 1.0
  (*@ p = example1 d
    requires
      is_empty (!st) /\
      sampled d t_n /\
      d.scale = Interval /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u
    ensures
      Eq p = compose_pvs fmlA !st &&
```

```
Welcome to Why3 IDE
type 'help' for help

File "/Users/ken/.opam/5.0.0/share/why3/stdlib/statELHT.mlw", line 17, ch

File "/Users/ken/.opam/5.0.0/share/why3/stdlib/atomicFormulas.mlw", line

File "/Users/ken/.opam/5.0.0/share/why3/stdlib/atomicFormulas.mlw", line
```

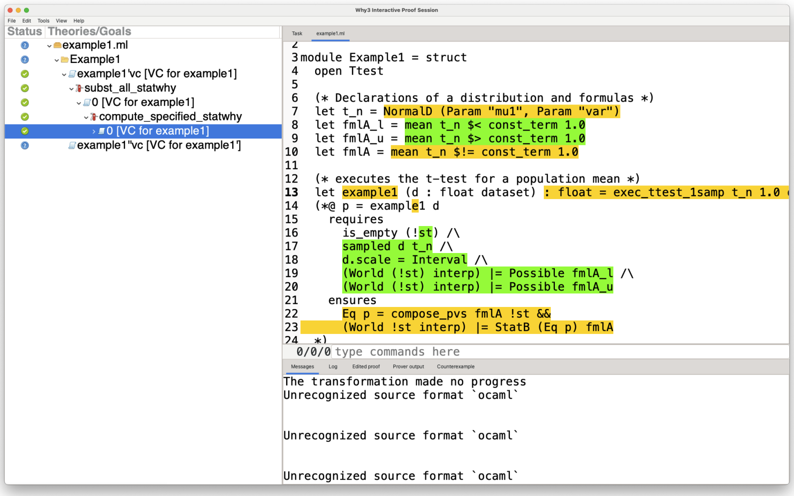
The Why3 IDE screen displays the code to be verified in the right panel and shows the VCs generated by the code in the left panel. Why3 generates two VCs from the source code file `example1.ml` in this case, `example1'vc` and `example1''vc`. The former is a VC for `example1`, which was explained in Section 4. The latter VC is similar to the former, but lacks one of the preconditions: `sampled d t_n`.

To discharge each VC, right-click on the VC, click “StatWhy” or press ‘4’ after selecting the VC. Then StatWhy starts discharging the VC.



```
example1.ml
  Example1
    example1'vc [VC for example1]
    example1''vc [VC for exam
```

If a prover succeeds in discharging a goal, a check-mark (✓) will appear to the left of the VC, indicating that the goal is correct:



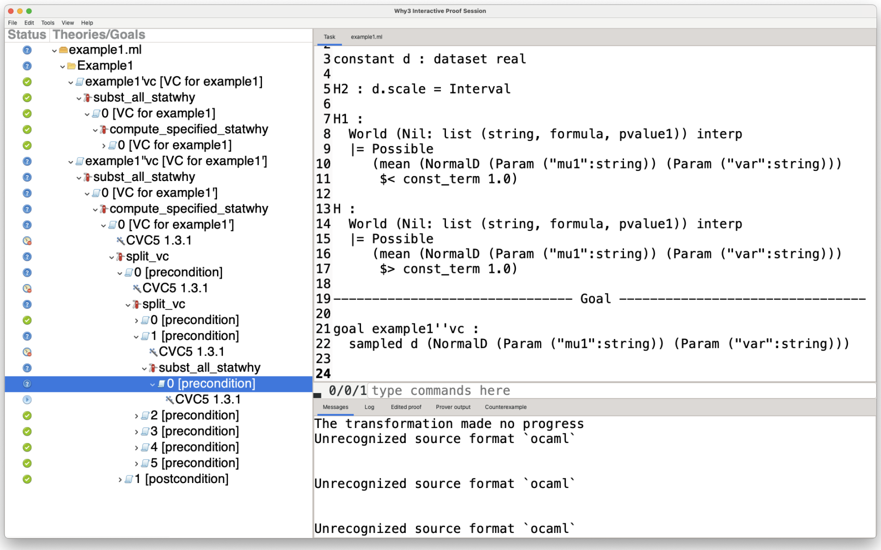
```
example1.ml
  Example1
    example1'vc [VC for example1]
      subst_all_statwhy
        0 [VC for example1]
          compute_specified_statwhy
            0 [VC for example1]
    example1''vc [VC for example1']
```

```
module Example1 = struct
  open Ttest

  (* Declarations of a distribution and formulas *)
  let t_n = NormalD (Param "mu1", Param "var")
  let fmlA_l = mean t_n $< const_term 1.0
  let fmlA_u = mean t_n $> const_term 1.0
  let fmlA = mean t_n $!= const_term 1.0

  (* executes the t-test for a population mean *)
  let example1 (d : float dataset) : float = exec_ttest_1samp t_n 1.0
  (*@ p = example1 d
    requires
      is_empty (!st) /\
      sampled d t_n /\
      d.scale = Interval /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u
    ensures
      Eq p = compose_pvs fmlA !st &&
      (World !st interp) |= StatB (Eq p) fmlA
  *)
```

```
0/0/0 type commands here
```

```
The transformation made no progress
Unrecognized source format `ocaml`


Unrecognized source format `ocaml`


Unrecognized source format `ocaml`
```

If Why3 fails to discharge the goal, by clicking a failed VC, the analyst finds the judgment that cannot be discharged on the right panel. For example, according to the goal on the right panel of the following figure, StatWhy has failed to check that the dataset `d` has been sampled from a normal distribution `NormalD (Param "mu1", Param "var")` with an unknown mean `mu1` and an unknown variance `var`.



```
Status  Theories/Goals
?  example1.ml
?    Example1
✓      example1'vc [VC for example1]
✓        subst_all_statwhy
✓          0 [VC for example1]
✓            compute_specified_statwhy
✓              0 [VC for example1]
?      example1''vc [VC for example1']
?        subst_all_statwhy
?          0 [VC for example1']
?            compute_specified_statwhy
?              0 [VC for example1']
                 CVC5 1.3.1
?                split_vc
?                  0 [precondition]
                     CVC5 1.3.1
?                    split_vc
✓                      0 [precondition]
?                      1 [precondition]
                         CVC5 1.3.1
?                        subst_all_statwhy
?                          0 [precondition]
                             CVC5 1.3.1
✓                      2 [precondition]
✓                      3 [precondition]
✓                      4 [precondition]
✓                      5 [precondition]
✓                1 [postcondition]
```

```
constant d : dataset real

H2 : d.scale = Interval

H1 :
  World (Nil: list (string, formula, pvalue1)) interp
  |= Possible
     (mean (NormalD (Param ("mu1":string)) (Param ("var":string)))
      $< const_term 1.0)

H :
  World (Nil: list (string, formula, pvalue1)) interp
  |= Possible
     (mean (NormalD (Param ("mu1":string)) (Param ("var":string)))
      $> const_term 1.0)

------------------------------- Goal --------------------------------

goal example1''vc :
  sampled d (NormalD (Param ("mu1":string)) (Param ("var":string)))


0/0/1 type commands here
```

```
The transformation made no progress
Unrecognized source format `ocaml`


Unrecognized source format `ocaml`


Unrecognized source format `ocaml`
```

Notes on our Custom Proof Strategy "StatWhy" is our custom proof strategy. It first applies Why3's default proof transformations (e.g., `split_vc` for splitting conjunctive verification conditions into smaller ones and `compute _specified` for unfolding certain functions and predicates and simplifying the proof goals). These invocations of the proof strategies are interleaved with calls to SMT solvers, whose timeouts are set to small values. If these applications of the default proof strategies fail to discharge the VCs, then we apply aggressive transformations that unfold the definitions of the functions and predicates defined in StatWhy.

In contrast, "StatWhy aggressive" is a more eager unfolding strategy. It first unfolds the definitions of all functions and predicates used in the goals and simplifies them. Therefore, in most cases, it discharges the VCs faster than the "StatWhy" strategy. However, in some cases where one of the preconditions does not hold, it may transform the goals so eagerly that the original structure of goals is lost, and the programmer cannot see which condition fails to be discharged. To identify such failed conditions, we recommend using the "StatWhy" strategy.

6.2 Several Variants of t-tests

StatWhy can distinguish between the different preconditions required for different hypothesis testing commands and remind users to make such conditions explicit. In this example, we 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 t-test* is applied otherwise, e.g., when two datasets are sampled from the population independently. StatWhy can check which of the tests should be applied to the current situation by checking the precondition.

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

```
val exec_ttest_paired (d1 d2: distribution) (y1 y2 : dataset real) (alt :
    alternative) : real
writes { st }
requires {
  paired y1 y2 /\ scale_leq Interval y1.scale /\ scale_leq Interval y2.scale /\
  (World !st interp) |= is_normal d1 /\ (World !st interp) |= is_normal d2 /\
  sampled y1 d1 /\ sampled y2 d2 /\
  let r1 = mean d1 in
  let r2 = mean d2 in
  match alt with
  | Two ->
    (World !st interp) |= Possible (r1 $< r2) /\
    (World !st interp) |= Possible (r1 $> r2)
  | Up ->
    (World !st interp) |= Not (Possible (r1 $< r2)) /\
    (World !st interp) |= Possible (r1 $> r2)
  | Low ->
    (World !st interp) |= Possible (r1 $< r2) /\
    (World !st interp) |= Not (Possible (r1 $> r2))
  end
}
ensures {
  let pv = result in
  pvalue pv /\
  let r1 = mean d1 in
  let r2 = mean d2 in
  let h = match alt with
    | Two -> r1 $!= r2
    | Up -> r1 $> r2
    | Low -> r1 $< r2
  end in !st = Cons ("ttest_paired", h, Eq pv) !(old st)
}
```

`exec_ttest_paired` takes five arguments. `d1` and `d2` denote the population distributions, `y1` and `y2` denote the datasets to be tested, and `alt` reopresents what type of the 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 described in the `requires` clause. `paired y1 y2` specifies that two samples `y1` and `y2` are obtained in pairs. The `ensures` clause specifies the postcondition of the test. The expression `!st = Cons ("ttest_paired", h, Eq pv) !(old st)` stores the resulting p-value `Eq pv` and the alternative hypothesis `h` in the store `st`.

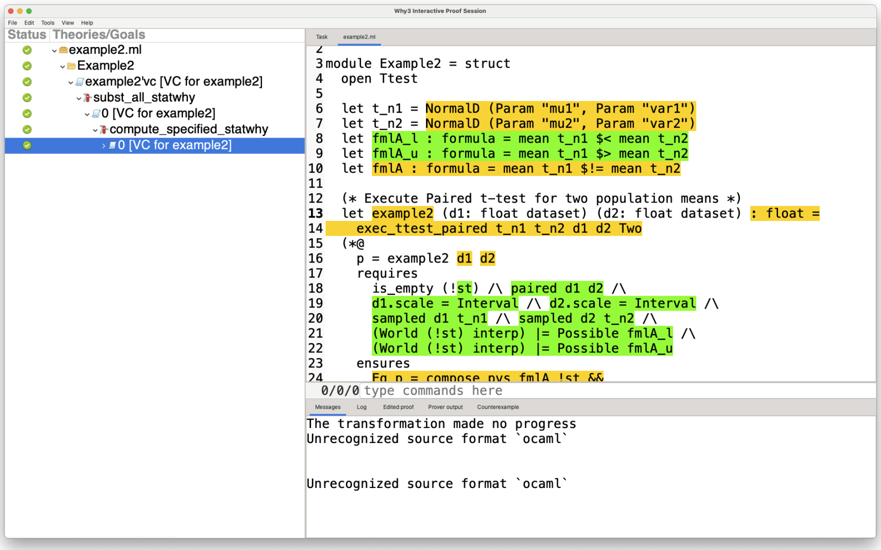
Non-Paired *t*-test The code below shows the specification of the non-paired *t*-test (Student's *t*-test):

```
val exec_ttest_ind_eq (d1 d2: distribution) (y1 y2 : dataset real) (alt :
    alternative) : real
writes { st }
requires {
  independent y1 y2 /\ scale_leq Interval y1.scale /\ scale_leq Interval y2.
      scale /\
  (World !st interp) |= is_normal d1 /\ (World !st interp) |= is_normal d2 /\
  (World !st interp) |= eq_variance d1 d2 /\
  (World !st interp) |= Not (check_variance d1) /\
  (World !st interp) |= Not (check_variance d2) /\
  sampled y1 d1 /\ sampled y2 d2 /\
  let r1 = mean d1 in
  let r2 = mean d2 in
  match alt with
  | Two ->
    (World !st interp) |= Possible (r1 $< r2) /\
    (World !st interp) |= Possible (r1 $> r2)
  | Up ->
    (World !st interp) |= Not (Possible (r1 $< r2)) /\
    (World !st interp) |= Possible (r1 $> r2)
  | Low ->
    (World !st interp) |= Possible (r1 $< r2) /\
    (World !st interp) |= Not (Possible (r1 $> r2))
  end
}
ensures {
  let pv = result in
  pvalue pv /\
  let r1 = mean d1 in
  let r2 = mean d2 in
  let h = match alt with
    | Two -> r1 $!= r2
    | Up -> r1 $> r2
    | Low -> r1 $< r2
  end in !st = Cons ("ttest_ind_eq", h, Eq pv) !(old st)
}
```

The precondition for `exec_ttest_ind_eq` includes the following conditions:

- `independent y1 y2` specifies that the datasets `y1` and `y2` must be sampled independently (not in pair).
- `(World !st interp) |= eq_variance d1 d2` specifies that the population distributions `d1` and `d2` have the same variance.
- `(World !st interp) |= Not (check_variance d1)` and `(World !st interp) |= Not (check_variance d2)` specify that the variances of `d1` and `d2` have not been checked, i.e., *unknown* to the programmer.

In the file `examples/example2.ml`, we have an example that conducts the paired *t*-test command `exec_ttest_paired`. To verify this code, execute `statwhy example2.ml`, which will open the Why3 IDE as follows.



```
module Example2 = struct
  open Ttest

  let t_n1 = NormalD (Param "mu1", Param "var1")
  let t_n2 = NormalD (Param "mu2", Param "var2")
  let fmlA_l : formula = mean t_n1 $< mean t_n2
  let fmlA_u : formula = mean t_n1 $> mean t_n2
  let fmlA : formula = mean t_n1 $!= mean t_n2

  (* Execute Paired t-test for two population means *)
  let example2 (d1: float dataset) (d2: float dataset) : float =
    exec_ttest_paired t_n1 t_n2 d1 d2 Two
  (*@
    p = example2 d1 d2
    requires
      is_empty (!st) /\ paired d1 d2 /\
      d1.scale = Interval /\ d2.scale = Interval /\
      sampled d1 t_n1 /\ sampled d2 t_n2 /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u
    ensures
      Eq p = compose pvs fmlA !st &&
```

```
The transformation made no progress
Unrecognized source format `ocaml`


Unrecognized source format `ocaml`
```

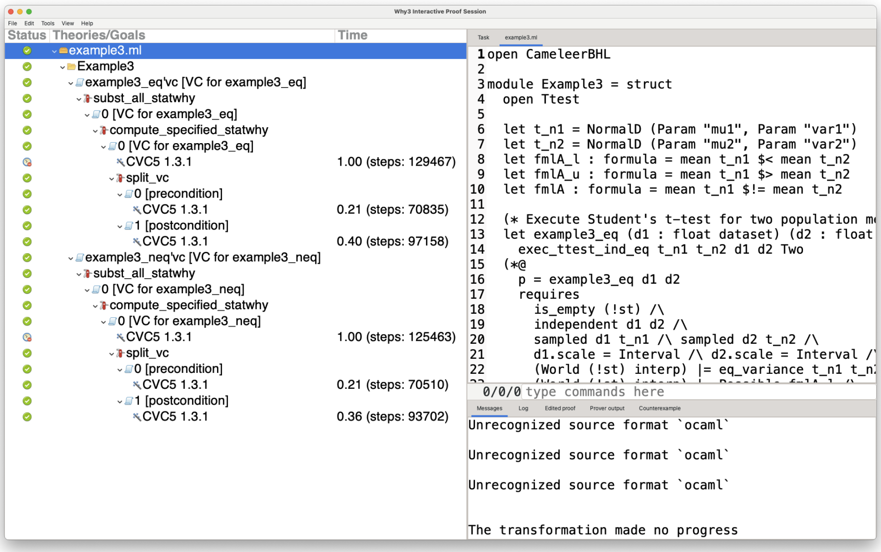
Equal vs. Unequal Variance We explain the *equal variance* assumption in the non-paired t-test as follows. In the non-paired t-test, we should use different t-tests according to this assumption. When we know or assume that two populations have equal variance, we usually use *Student's t-test*. In contrast, 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. `(World !st interp) |= eq_variance d1 d2` in the precondition represents that the two distributions `d1` and `d2` that respectively draw the datasets `y1` and `y2` have equal variance.

In contrast, `exec_ttest_ind_neq` below assumes `Not (eq_variance d1 d2)`:

```
val exec_ttest_ind_neq (d1 d2: distribution) (y1 y2 : dataset real) (alt :
     alternative) : real
writes { st }
requires {
  independent y1 y2 /\ scale_leq Interval y1.scale /\ scale_leq Interval y2.
       scale /\
  (World !st interp) |= is_normal d1 /\ (World !st interp) |= is_normal d2 /\
  (World !st interp) |= Not (check_variance d1) /\
  (World !st interp) |= Not (check_variance d2) /\
  (World !st interp) |= Not (eq_variance d1 d2) /\
  sampled y1 d1 /\ sampled y2 d2 /\
  let r1 = mean d1 in
  let r2 = mean d2 in
  match alt with
  | Two ->
    (World !st interp) |= Possible (r1 $< r2) /\
    (World !st interp) |= Possible (r1 $> r2)
  | Up ->
    (World !st interp) |= Not (Possible (r1 $< r2)) /\
    (World !st interp) |= Possible (r1 $> r2)
  | Low ->
    (World !st interp) |= Possible (r1 $< r2) /\
    (World !st interp) |= Not (Possible (r1 $> r2))
  end
}
ensures {
  let pv = result in
  pvalue pv /\
  let r1 = mean d1 in
  let r2 = mean d2 in
  let h = match alt with
    | Two -> r1 $!= r2
    | Up -> r1 $> r2
    | Low -> r1 $< r2
  end in !st = Cons ("ttest_ind_neq", h, Eq pv) !(old st)
}
```

The code examples for these t-tests are available in `examples/example3.ml`


Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Time
Task example3.ml
Messages Log Edited proof Prover output Counterexample

```
example3.ml
  Example3
    example3_eq'vc [VC for example3_eq]
      subst_all_statwhy
        0 [VC for example3_eq]
          compute_specified_statwhy
            0 [VC for example3_eq]
              CVC5 1.3.1                        1.00 (steps: 129467)
              split_vc
                0 [precondition]
                  CVC5 1.3.1                    0.21 (steps: 70835)
                1 [postcondition]
                  CVC5 1.3.1                    0.40 (steps: 97158)
    example3_neq'vc [VC for example3_neq]
      subst_all_statwhy
        0 [VC for example3_neq]
          compute_specified_statwhy
            0 [VC for example3_neq]
              CVC5 1.3.1                        1.00 (steps: 125463)
              split_vc
                0 [precondition]
                  CVC5 1.3.1                    0.21 (steps: 70510)
                1 [postcondition]
                  CVC5 1.3.1                    0.36 (steps: 93702)
```

```
open CameleerBHL

module Example3 = struct
  open Ttest

  let t_n1 = NormalD (Param "mu1", Param "var1")
  let t_n2 = NormalD (Param "mu2", Param "var2")
  let fmlA_l : formula = mean t_n1 $< mean t_n2
  let fmlA_u : formula = mean t_n1 $> mean t_n2
  let fmlA : formula = mean t_n1 $!= mean t_n2

  (* Execute Student's t-test for two population m
  let example3_eq (d1 : float dataset) (d2 : float
    exec_ttest_ind_eq t_n1 t_n2 d1 d2 Two
  (*@
    p = example3_eq d1 d2
    requires
      is_empty (!st) /\
      independent d1 d2 /\
      sampled d1 t_n1 /\ sampled d2 t_n2 /\
      d1.scale = Interval /\ d2.scale = Interval /
      (World (!st) interp) |= eq_variance t_n1 t_n
```

```
0/0/0 type commands here
```

```
Unrecognized source format `ocaml`

Unrecognized source format `ocaml`

Unrecognized source format `ocaml`


The transformation made no progress
```

6.3 Dealing with Combined Tests in StatWhy

Let A_{φ_0} and A_{φ_1} be two hypothesis tests with alternative hypotheses φ_0 and φ_1, respectively. There are two possible combinations of A_{φ_0} and A_{φ_1}. One is the disjunctive combination $A_{\varphi_0 \vee \varphi_1}$ whose alternative hypothesis is $\varphi_0 \vee \varphi_1$, while the other is the conjunctive combination $A_{\varphi_0 \wedge \varphi_1}$ with alternative hypothesis $\varphi_0 \wedge \varphi_1$. StatWhy can check whether a program correctly calculates the p-value of such combined tests.

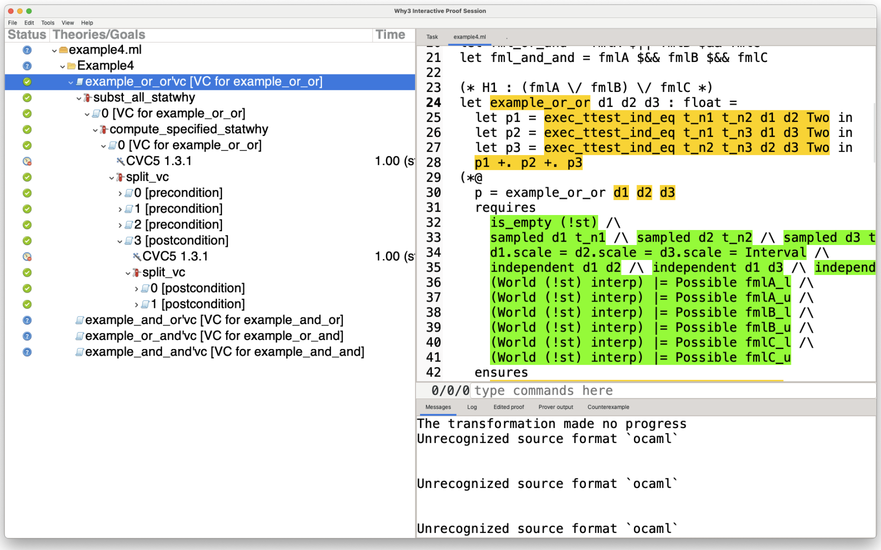
***P*-Values of Disjunctive Alternative Hypothesis** Assume that p-values of A_{φ_0} and A_{φ_1} are p_0 and p_1, respectively. It is known that the p-value p of $A_{\varphi_0 \vee \varphi_1}$ satisfies $p \leq p_0 + p_1$, which is called the Bonferroni correction. StatWhy automatically calculates the p-value of $A_{\varphi_0 \wedge \varphi_1}$ as $p_0 + p_1$ if A_{φ_0} and A_{φ_1} are performed independently.

The code below defines a procedure `example_or_or`, which compares the dataset `d1` with `d2` and `d3`, and `d2` with `d3`, then calculates the overall p-value.

examples/example4.ml

```
module Example4 = struct
  open Ttest

  let t_n1 = NormalD (Param "mu1", Param "var")
  let t_n2 = NormalD (Param "mu2", Param "var")
  let t_n3 = NormalD (Param "mu3", Param "var")
  let fmlA_l = mean t_n1 $< mean t_n2
  let fmlA_u = mean t_n1 $> mean t_n2
  let fmlA = mean t_n1 $!= mean t_n2
  let fmlB_l = mean t_n1 $< mean t_n3
  let fmlB_u = mean t_n1 $> mean t_n3
  let fmlB = mean t_n1 $!= mean t_n3
  let fmlC_l = mean t_n2 $< mean t_n3
  let fmlC_u = mean t_n2 $> mean t_n3
  let fmlC = mean t_n2 $!= mean t_n3
  let fml_or_or = fmlA $|| fmlB $|| fmlC
  let fml_and_or = fmlA $&& fmlB $|| fmlC
  let fml_or_and = fmlA $|| fmlB $&& fmlC
  let fml_and_and = fmlA $&& fmlB $&& fmlC

  (* H1 : (fmlA \/ fmlB) \/ fmlC *)
  let example_or_or d1 d2 d3 : float =
    let p1 = exec_ttest_ind_eq t_n1 t_n2 d1 d2 Two in
    let p2 = exec_ttest_ind_eq t_n1 t_n3 d1 d3 Two in
    let p3 = exec_ttest_ind_eq t_n2 t_n3 d2 d3 Two in
    p1 +. p2 +. p3
  (*@
    p = example_or_or d1 d2 d3
    requires
      is_empty (!st) /\
      sampled d1 t_n1 /\ sampled d2 t_n2 /\ sampled d3 t_n3 /\
      d1.scale = d2.scale = d3.scale = Interval /\
      independent d1 d2 /\ independent d1 d3 /\ independent d2 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 /\
      (World (!st) interp) |= Possible fmlC_l /\
      (World (!st) interp) |= Possible fmlC_u
    ensures
      (Leq p) = compose_pvs fml_or_or !st &&
      (World !st interp) |= StatB (Leq p) ((((mean t_n1) $!= (mean t_n2)) $||
           fmlB) $|| fmlC)
  *)
  ...
end
```

To verify this code, run `statwhy examples/example4.ml` and apply the "StatWhy" strategy to `example_or_or'vc`.



Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Time
Task example4.ml
Messages Log Edited proof Prover output Counterexample

```
example4.ml
  Example4
    example_or_or'vc [VC for example_or_or]
      subst_all_statwhy
        0 [VC for example_or_or]
          compute_specified_statwhy
            0 [VC for example_or_or]
              CVC5 1.3.1                                1.00 (s
              split_vc
                0 [precondition]
                1 [precondition]
                2 [precondition]
                3 [postcondition]
                  CVC5 1.3.1                            1.00 (s
                  split_vc
                    0 [postcondition]
                    1 [postcondition]
      example_and_or'vc [VC for example_and_or]
      example_or_and'vc [VC for example_or_and]
      example_and_and'vc [VC for example_and_and]
```

```
let fml_and_and = fmlA $&& fmlB $&& fmlC

(* H1 : (fmlA \/ fmlB) \/ fmlC *)
let example_or_or d1 d2 d3 : float =
  let p1 = exec_ttest_ind_eq t_n1 t_n2 d1 d2 Two in
  let p2 = exec_ttest_ind_eq t_n1 t_n3 d1 d3 Two in
  let p3 = exec_ttest_ind_eq t_n2 t_n3 d2 d3 Two in
  p1 +. p2 +. p3
(*@
  p = example_or_or d1 d2 d3
  requires
    is_empty (!st) /\
    sampled d1 t_n1 /\ sampled d2 t_n2 /\ sampled d3 t
    d1.scale = d2.scale = d3.scale = Interval /\
    independent d1 d2 /\ independent d1 d3 /\ independ
    (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
```

```
0/0/0 type commands here
```

```
The transformation made no progress
Unrecognized source format `ocaml`


Unrecognized source format `ocaml`


Unrecognized source format `ocaml`
```

***P*-Values of Conjunctive Hypotheses** To calculate the p-value of a conjunctive hypothesis (e.g., $\varphi_0 \wedge \varphi_1$), we take the minimum of the p-values of its subformulas φ_0 and φ_1. StatWhy can also verify these conjunctive combinations of hypothesis tests.

The code below compares `d1`, `d2`, and `d3`, as the code in the last section, but reports the smallest p-value among the three comparisons.

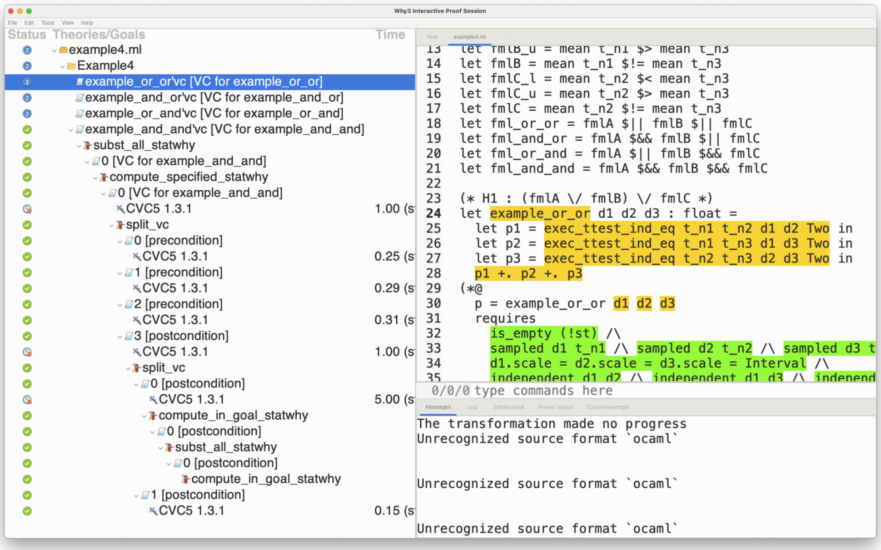
examples/example4.ml

```
open CameleerBHL

module Example4 = struct
  open Ttest

  let t_n1 = NormalD (Param "mu1", Param "var")
  let t_n2 = NormalD (Param "mu2", Param "var")
  let t_n3 = NormalD (Param "mu3", Param "var")
  let fmlA_l = mean t_n1 $< mean t_n2
  let fmlA_u = mean t_n1 $> mean t_n2
  let fmlA = mean t_n1 $!= mean t_n2
  let fmlB_l = mean t_n1 $< mean t_n3
  let fmlB_u = mean t_n1 $> mean t_n3
  let fmlB = mean t_n1 $!= mean t_n3
  let fmlC_l = mean t_n2 $< mean t_n3
  let fmlC_u = mean t_n2 $> mean t_n3
  let fmlC = mean t_n2 $!= mean t_n3
  let fml_or_or = fmlA $|| fmlB $|| fmlC
  let fml_and_or = fmlA $&& fmlB $|| fmlC
  let fml_or_and = fmlA $|| fmlB $&& fmlC
  let fml_and_and = fmlA $&& fmlB $&& fmlC

  ...

  (* H1 : (fmlA /\ fmlB) /\ fmlC *)
  let example_and_and d1 d2 d3 : float =
    let p1 = exec_ttest_ind_eq t_n1 t_n2 d1 d2 Two in
    let p2 = exec_ttest_ind_eq t_n1 t_n3 d1 d3 Two in
    let p3 = exec_ttest_ind_eq t_n2 t_n3 d2 d3 Two in
    min (min p1 p2) p3
  (*@
    p = example_and_and d1 d2 d3
    requires
      is_empty (!st) /\
      sampled d1 t_n1 /\ sampled d2 t_n2 /\ sampled d3 t_n3 /\
      d1.scale = d2.scale = d3.scale = Interval /\
      independent d1 d2 /\ independent d1 d3 /\ independent d2 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 /\
      (World (!st) interp) |= Possible fmlC_l /\
      (World (!st) interp) |= Possible fmlC_u
    ensures
      (Leq p) = compose_pvs fml_and_and !st &&
      (World !st interp) |= StatB (Leq p) fml_and_and
  *)
end
```

To verify the code above, execute the following command:

```
$ statwhy examples/example4.ml
```

And apply the “StatWhy” strategy to `example_and_and'vc`.



```
Status Theories/Goals                                        Time
?  example4.ml
?    Example4
?      example_or_or'vc [VC for example_or_or]
?      example_and_or'vc [VC for example_and_or]
?      example_or_and'vc [VC for example_or_and]
✓      example_and_and'vc [VC for example_and_and]
✓        subst_all_statwhy
✓          0 [VC for example_and_and]
✓            compute_specified_statwhy
✓              0 [VC for example_and_and]
                 CVC5 1.3.1                                  1.00 (s
✓                split_vc
✓                  0 [precondition]
✓                    CVC5 1.3.1                              0.25 (s
✓                  1 [precondition]
✓                    CVC5 1.3.1                              0.29 (s
✓                  2 [precondition]
✓                    CVC5 1.3.1                              0.31 (s
✓                  3 [postcondition]
                     CVC5 1.3.1                              1.00 (s
✓                    split_vc
✓                      0 [postcondition]
                         CVC5 1.3.1                          5.00 (s
✓                        compute_in_goal_statwhy
✓                          0 [postcondition]
✓                            subst_all_statwhy
✓                              0 [postcondition]
✓                                compute_in_goal_statwhy
✓                      1 [postcondition]
✓                        CVC5 1.3.1                          0.15 (s
```

```
let fmlB_u = mean t_n1 $> mean t_n3
let fmlB = mean t_n1 $!= mean t_n3
let fmlC_l = mean t_n2 $< mean t_n3
let fmlC_u = mean t_n2 $> mean t_n3
let fmlC = mean t_n2 $!= mean t_n3
let fml_or_or = fmlA $|| fmlB $|| fmlC
let fml_and_or = fmlA $&& fmlB $|| fmlC
let fml_or_and = fmlA $|| fmlB $&& fmlC
let fml_and_and = fmlA $&& fmlB $&& fmlC

(* H1 : (fmlA \/ fmlB) \/ fmlC *)
let example_or_or d1 d2 d3 : float =
  let p1 = exec_ttest_ind_eq t_n1 t_n2 d1 d2 Two in
  let p2 = exec_ttest_ind_eq t_n1 t_n3 d1 d3 Two in
  let p3 = exec_ttest_ind_eq t_n2 t_n3 d2 d3 Two in
  p1 +. p2 +. p3
(*@
  p = example_or_or d1 d2 d3
  requires
    is_empty (!st) /\
    sampled d1 t_n1 /\ sampled d2 t_n2 /\ sampled d3 t
    d1.scale = d2.scale = d3.scale = Interval /\
    independent d1 d2 /\ independent d1 d3 /\ independ
0/0/0 type commands here
```

```
The transformation made no progress
Unrecognized source format `ocaml`


Unrecognized source format `ocaml`


Unrecognized source format `ocaml`
```

***P*-Value Hacking** The *p-value hacking* or *data dredging* is a method for manipulating statistical analysis to obtain a lower p-value than the actual one. In this example, we see that StatWhy prevents the p-value hacking by calculating correct p-values with the results of conducted hypothesis tests.

The following code, `example5` in `examples/example5.ml`, is an example of p-value hacking:

examples/example5.ml

```
open CameleerBHL

module Example5 = struct
  open Ttest

  let t_n = NormalD (Param "mu1", Param "var")
  let fmlA_l = mean t_n $< const_term 1.0
  let fmlA_u = mean t_n $> const_term 1.0
  let fmlA = mean t_n $!= const_term 1.0

  (* Example of p-value hacking *)
  (* This program is INCORRECT and so its verification FAILS *)
  let example5 d1 d2 =
    let p1 = exec_ttest_1samp t_n 1.0 d1 Two in
    let p2 = exec_ttest_1samp t_n 1.0 d2 Two in
    let p = min p1 p2 in
    (p1, p2, p)
  (*@
    (p1, p2, p) = ex_hack d1 d2
    requires
      is_empty (!st) /\
      sampled d1 t_n /\ sampled d2 t_n /\
      d1.scale = d2.scale = Interval /\
      (World (!st) interp) |= Possible fmlA_l /\
      (World (!st) interp) |= Possible fmlA_u
    ensures
      (Eq p = compose_pvs fmlA !st (* This is incorrect *)
        && (World !st interp) |= StatB (Eq p) fmlA) /\
      (Leq (p1 +. p2) = compose_pvs fmlA !st (* This is correct *)
        && (World !st interp) |= StatB (Leq (p1 +. p2)) fmlA)
  *)
end
```

`example5` performs the *t*-test for the mean of `t_n` twice, using different datasets `d1` and `d2`. Then it obtains the *p*-values for each test, `p1` and `p2`, and reports the lower *p*-value `p` (defined by `min p1 p2`).

The postcondition of this function consists of two main formulas:

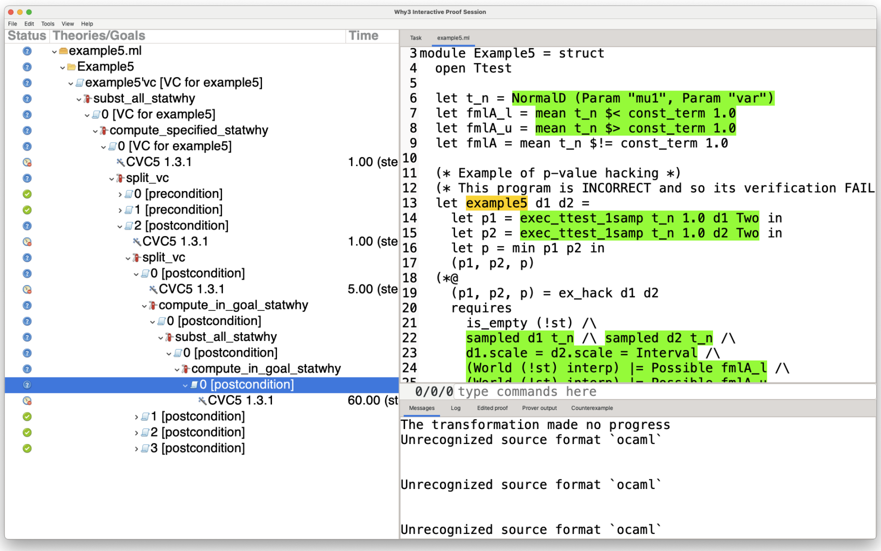
```
Eq p = compose_pvs fmlA !st && (World !st interp) |= StatB (Eq p) fmlA
```

and

```
Leq (p1 +. p2) = compose_pvs fmlA !st
&& (World !st interp) |= StatB (Leq (p1 +. p2)) fmlA.
```

In the former assertion, `Eq p = compose_pvs fmlA !st` is a wrong interpretation of the result. In contrast, the latter states that the sum of `p1` and `p2` is the *p*-value of `fmlA`, which is correct.

StatWhy does validate the latter, but not the former; `Eq p = compose_pvs fmlA !st` fails to be validated:



Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Time
Task example5.ml
Messages Log Edited proof Prover output Counterexample

```
example5.ml
  Example5
    example5'vc [VC for example5]
      subst_all_statwhy
        0 [VC for example5]
          compute_specified_statwhy
            0 [VC for example5]
              CVC5 1.3.1                                   1.00 (ste
              split_vc
                0 [precondition]
                1 [precondition]
                2 [postcondition]
                  CVC5 1.3.1                               1.00 (ste
                  split_vc
                    0 [postcondition]
                      CVC5 1.3.1                           5.00 (ste
                      compute_in_goal_statwhy
                        0 [postcondition]
                          subst_all_statwhy
                            0 [postcondition]
                              compute_in_goal_statwhy
                                0 [postcondition]
                                  CVC5 1.3.1               60.00 (st
                    1 [postcondition]
                    2 [postcondition]
                    3 [postcondition]
```

```
module Example5 = struct
  open Ttest

  let t_n = NormalD (Param "mu1", Param "var")
  let fmlA_l = mean t_n $< const_term 1.0
  let fmlA_u = mean t_n $> const_term 1.0
  let fmlA = mean t_n $!= const_term 1.0

  (* Example of p-value hacking *)
  (* This program is INCORRECT and so its verification FAIL
  let example5 d1 d2 =
    let p1 = exec_ttest_1samp t_n 1.0 d1 Two in
    let p2 = exec_ttest_1samp t_n 1.0 d2 Two in
    let p = min p1 p2 in
    (p1, p2, p)
  (*@
    (p1, p2, p) = ex_hack d1 d2
    requires
      is_empty (!st) /\
      sampled d1 t_n /\ sampled d2 t_n /\
      d1.scale = d2.scale = Interval /\
      (World (!st) interp) |= Possible fmlA_l /\
```

```
0/0/0 type commands here
```

```
The transformation made no progress
Unrecognized source format `ocaml`


Unrecognized source format `ocaml`


Unrecognized source format `ocaml`
```

6.4 Hypothesis Tests for More Than Two Population Means

This example illustrates hypothesis tests to analyze the difference among more than two population means.

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

These methods test whether all the given population means are identical or not. To test the difference of each pair of the given means, we should use a *multiple comparison method*, such as *Tukey's HSD test* (see Section 6.5.)

One-Way ANOVA The specification of the *one-way ANOVA* in StatWhy is as follows:

```
val exec_oneway_ANOVA (ds : list distribution) (ys : list (dataset real)) :
     real
writes { st }
requires {
  independent_list ys /\
  length ds = length ys &&
  length ds >= 2 /\ length ys >= 2 &&
  for_all2 (fun d y -> ((World !st interp) |= is_normal d) && sampled y d) ds
      ys /\
  for_all (fun y -> scale_leq Interval y.scale) ys /\
  (forall p q : distribution. mem p ds /\ mem q ds ->
    (World !st interp) |= eq_variance p q) /\
  (forall s t : distribution. mem s ds /\ mem t ds /\ not eq_distribution s t
      ->
    ((World !st interp) |= Possible (mean s $< mean t) /\
     (World !st interp) |= Possible (mean s $> mean t)))
} ensures {
  let pv = result in
  pvalue result /\
  let h = oneway_ANOVA_hypothesis ds in
  !st = !(old st) ++ Cons ("oneway_ANOVA", h, Eq 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:

```
for_all2
  (fun p y -> match p with | NormalD _ _ -> sampled y p | _ -> false end)
  ds ys
```

The latter is specified by:

```
  (forall p q : distribution. mem p ps /\ mem q ps /\ not eq_distribution s t ->
    (World !st interp) |= eq_variance p q)
```

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` applied to `Cons termA (Cons termB (Cons termC Nil))` represents the formula ($\texttt{termA} \neq \texttt{termB}$) $\wedge$ ($\texttt{termA} \neq \texttt{termC}$) $\wedge$ ($\texttt{termB} \neq \texttt{termC}$).

The following code conducts the one-way ANOVA for three population means.

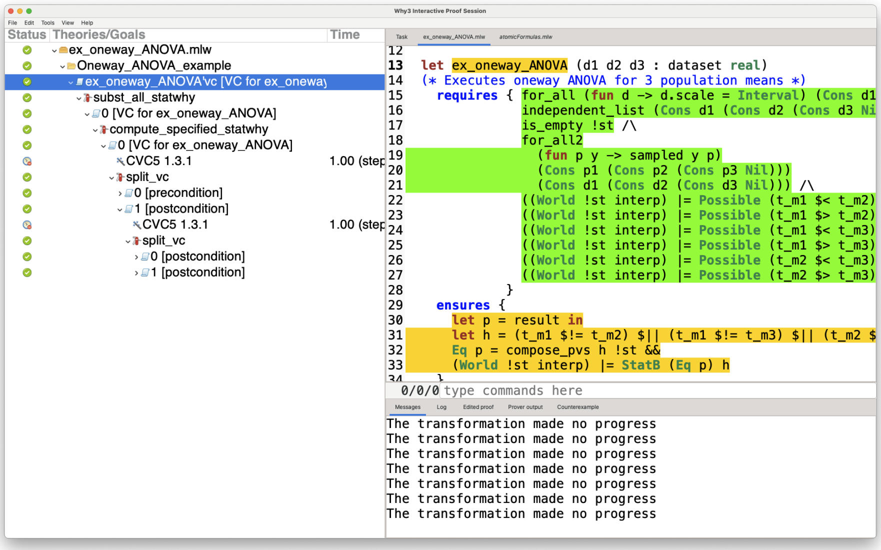
examples/mlw/ex_oneway_ANOVA.mlw

```
module Oneway_ANOVA_example
  use cameleerBHL.CameleerBHL
  use oneway_ANOVA.Oneway_ANOVA

  let function p1 = NormalD (Param "m1") (Param "v")
  let function p2 = NormalD (Param "m2") (Param "v")
  let function p3 = NormalD (Param "m3") (Param "v")

  let function t_m1 = mean p1
  let function t_m2 = mean p2
  let function t_m3 = mean p3

  let ex_oneway_ANOVA (d1 d2 d3 : dataset real)
  (* Executes oneway ANOVA for 3 population means *)
    requires { for_all (fun d -> d.scale = Interval) (Cons d1 (Cons d2 (Cons d3
         Nil))) /\
              independent_list (Cons d1 (Cons d2 (Cons d3 Nil))) /\
              is_empty !st /\
              for_all2
                (fun p y -> sampled y p)
                (Cons p1 (Cons p2 (Cons p3 Nil)))
                (Cons d1 (Cons d2 (Cons d3 Nil))) /\
              ((World !st interp) |= Possible (t_m1 $< t_m2)) /\
              ((World !st interp) |= Possible (t_m1 $> t_m2)) /\
              ((World !st interp) |= Possible (t_m1 $< t_m3)) /\
              ((World !st interp) |= Possible (t_m1 $> t_m3)) /\
              ((World !st interp) |= Possible (t_m2 $< t_m3)) /\
              ((World !st interp) |= Possible (t_m2 $> t_m3))
            }
    ensures {
      let p = result in
      let h = (t_m1 $!= t_m2) $|| (t_m1 $!= t_m3) $|| (t_m2 $!= t_m3) in
      Eq p = compose_pvs h !st &&
      (World !st interp) |= StatB (Eq p) h
    }
    = exec_oneway_ANOVA (Cons p1 (Cons p2 (Cons p3 Nil))) (Cons d1 (Cons d2 (Cons
         d3 Nil)))
end
```

In the postcondition, `h = fmlA $&& fmlB $&& fmlC` confirms the concrete form of the alternative hypothesis `h`.[12]

To verify the code above, execute the following command:

```
$ statwhy examples/mlw/ex_oneway_ANOVA.mlw
```


Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Time
Task ex_oneway_ANOVA.mlw atomicFormulas.mlw
Messages Log Edited proof Prover output Counterexample

```
ex_oneway_ANOVA.mlw
  Oneway_ANOVA_example
    ex_oneway_ANOVA'vc [VC for ex_oneway
      subst_all_statwhy
        0 [VC for ex_oneway_ANOVA]
          compute_specified_statwhy
            0 [VC for ex_oneway_ANOVA]
              CVC5 1.3.1                        1.00 (step
              split_vc
                0 [precondition]
                1 [postcondition]
                  CVC5 1.3.1                    1.00 (step
                  split_vc
                    0 [postcondition]
                    1 [postcondition]
```

```

let ex_oneway_ANOVA (d1 d2 d3 : dataset real)
(* Executes oneway ANOVA for 3 population means *)
  requires { for_all (fun d -> d.scale = Interval) (Cons d1
             independent_list (Cons d1 (Cons d2 (Cons d3 Ni
             is_empty !st /\
             for_all2
               (fun p y -> sampled y p)
               (Cons p1 (Cons p2 (Cons p3 Nil)))
               (Cons d1 (Cons d2 (Cons d3 Nil))) /\
             ((World !st interp) |= Possible (t_m1 $< t_m2)
             ((World !st interp) |= Possible (t_m1 $> t_m2)
             ((World !st interp) |= Possible (t_m1 $< t_m3)
             ((World !st interp) |= Possible (t_m1 $> t_m3)
             ((World !st interp) |= Possible (t_m2 $< t_m3)
             ((World !st interp) |= Possible (t_m2 $> t_m3)
           }
  ensures {
    let p = result in
    let h = (t_m1 $!= t_m2) $|| (t_m1 $!= t_m3) $|| (t_m2 $
    Eq p = compose_pvs h !st &&
    (World !st interp) |= StatB (Eq p) h
```

```
0/0/0 type commands here
```

```
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
```

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:

examples/mlw/ex_alexandergovern.mlw

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

  let function p1 : distribution = NormalD (Param "mean1") (Param "var1")
  let function p2 : distribution = NormalD (Param "mean2") (Param "var2")
  let function p3 : distribution = NormalD (Param "mean3") (Param "var3")
  let function p4 : distribution = NormalD (Param "mean4") (Param "var4")

  let function t_m1 = mean p1
  let function t_m2 = mean p2
  let function t_m3 = mean p3
  let function t_m4 = mean p4
```

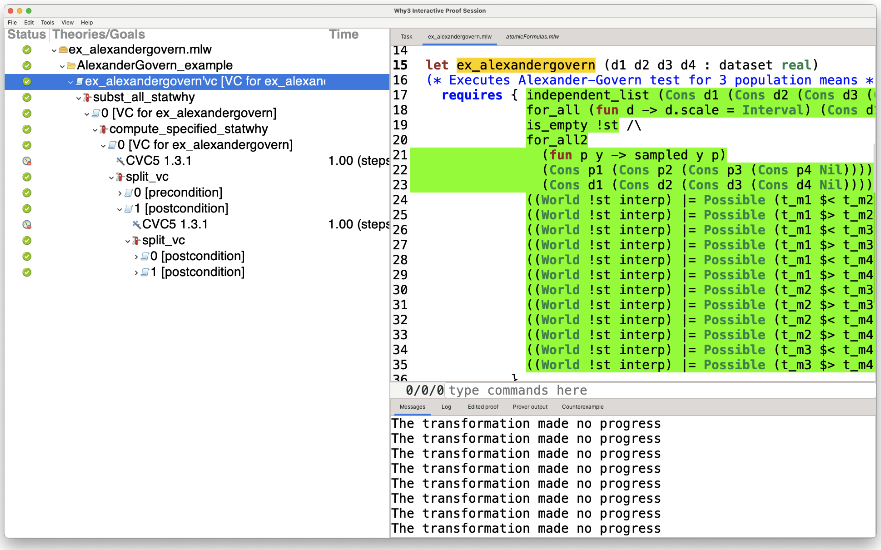
[12] `$&&` is the syntax sugar for the conjunction of two hypotheses. Similarly, `$||` is for the disjunction of two hypotheses.

```
  let ex_alexandergovern (d1 d2 d3 d4 : dataset real)
  (* Executes Alexander-Govern test for 3 population means *)
    requires { independent_list (Cons d1 (Cons d2 (Cons d3 (Cons d4 Nil)))) /\
               for_all (fun d -> d.scale = Interval) (Cons d1 (Cons d2 (Cons d3 (
                    Cons d4 Nil)))) /\
               is_empty !st /\
               for_all2
                 (fun p y -> sampled y p)
                 (Cons p1 (Cons p2 (Cons p3 (Cons p4 Nil))))
                 (Cons d1 (Cons d2 (Cons d3 (Cons d4 Nil)))) /\
               ((World !st interp) |= Possible (t_m1 $< t_m2)) /\
               ((World !st interp) |= Possible (t_m1 $> t_m2)) /\
               ((World !st interp) |= Possible (t_m1 $< t_m3)) /\
               ((World !st interp) |= Possible (t_m1 $> t_m3)) /\
               ((World !st interp) |= Possible (t_m1 $< t_m4)) /\
               ((World !st interp) |= Possible (t_m1 $> t_m4)) /\
               ((World !st interp) |= Possible (t_m2 $< t_m3)) /\
               ((World !st interp) |= Possible (t_m2 $> t_m3)) /\
               ((World !st interp) |= Possible (t_m2 $< t_m4)) /\
               ((World !st interp) |= Possible (t_m2 $> t_m4)) /\
               ((World !st interp) |= Possible (t_m3 $< t_m4)) /\
               ((World !st interp) |= Possible (t_m3 $> t_m4))
             }
    ensures {
      let p = result in
      let h = (t_m1 $!= t_m2) $|| (t_m1 $!= t_m3) $|| (t_m1 $!= t_m4)
        $|| (t_m2 $!= t_m3) $|| (t_m2 $!= t_m4) $|| (t_m3 $!= t_m4) in
      Eq p = compose_pvs h !st &&
      (World !st interp) |= StatB (Eq p) h
    }
    = exec_alexandergovern (Cons p1 (Cons p2 (Cons p3 (Cons p4 Nil)))) (Cons d1 (
        Cons d2 (Cons d3 (Cons d4 Nil))))
end
```

Note that the assumption of equal variance is no longer necessary in the precondition.

To verify the code above, execute the following command:

```
$ statwhy examples/mlw/ex_alexandergovern.mlw
```


Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Time
Task ex_alexandergovern.mlw atomicFormulas.mlw
Messages Log Edited proof Prover output Counterexample

```
ex_alexandergovern.mlw
  AlexanderGovern_example
    ex_alexandergovern'vc [VC for ex_alexan
      subst_all_statwhy
        0 [VC for ex_alexandergovern]
          compute_specified_statwhy
            0 [VC for ex_alexandergovern]
              CVC5 1.3.1                                  1.00 (steps
              split_vc
                0 [precondition]
                1 [postcondition]
                  CVC5 1.3.1                              1.00 (steps
                  split_vc
                    0 [postcondition]
                    1 [postcondition]
```

```

let ex_alexandergovern (d1 d2 d3 d4 : dataset real)
(* Executes Alexander-Govern test for 3 population means *
  requires { independent_list (Cons d1 (Cons d2 (Cons d3 (
             for_all (fun d -> d.scale = Interval) (Cons d
             is_empty !st /\
             for_all2
               (fun p y -> sampled y p)
               (Cons p1 (Cons p2 (Cons p3 (Cons p4 Nil))))
               (Cons d1 (Cons d2 (Cons d3 (Cons d4 Nil))))
             ((World !st interp) |= Possible (t_m1 $< t_m2
             ((World !st interp) |= Possible (t_m1 $> t_m2
             ((World !st interp) |= Possible (t_m1 $< t_m3
             ((World !st interp) |= Possible (t_m1 $> t_m3
             ((World !st interp) |= Possible (t_m1 $< t_m4
             ((World !st interp) |= Possible (t_m1 $> t_m4
             ((World !st interp) |= Possible (t_m2 $< t_m3
             ((World !st interp) |= Possible (t_m2 $> t_m3
             ((World !st interp) |= Possible (t_m2 $< t_m4
             ((World !st interp) |= Possible (t_m2 $> t_m4
             ((World !st interp) |= Possible (t_m3 $< t_m4
             ((World !st interp) |= Possible (t_m3 $> t_m4
```

```
0/0/0 type commands here
```

```
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
```

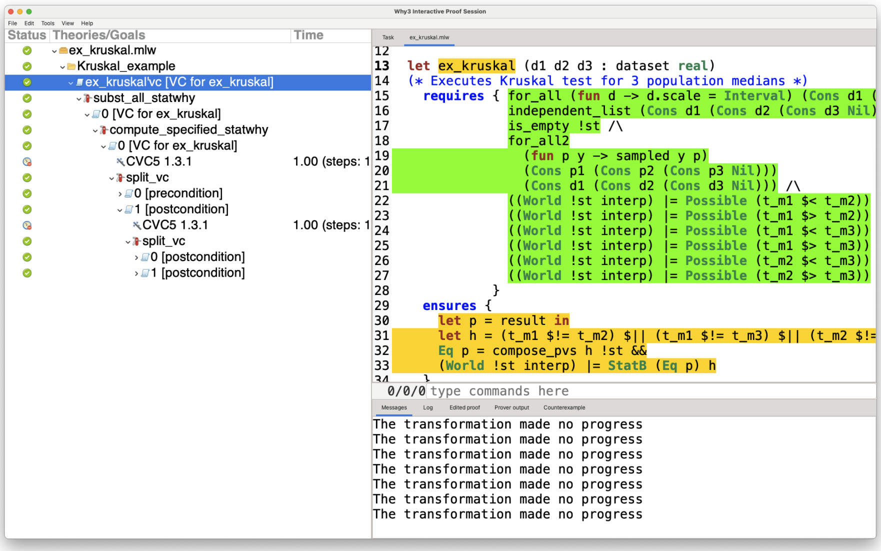
Kruskal-Wallis Test The *Kruskal-Wallis H-test* is a non-parametric variant of ANOVA; it does not assume that each sample is from normally distributed populations with equal variance. Here is an example of the Kruskal-Wallis H-test whose null hypothesis is that all medians of three populations are equal:

examples/mlw/ex_kruskal.mlw

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

  let function p1 = UnknownD "p1"
  let function p2 = UnknownD "p2"
  let function p3 = UnknownD "p3"

  let function t_m1 = med p1
  let function t_m2 = med p2
  let function t_m3 = med p3

  let ex_kruskal (d1 d2 d3 : dataset real)
  (* Executes Kruskal test for 3 population medians *)
    requires { for_all (fun d -> d.scale = Interval) (Cons d1 (Cons d2 (Cons d3
         Nil))) /\
               independent_list (Cons d1 (Cons d2 (Cons d3 Nil))) /\
               is_empty !st /\
               for_all2
                 (fun p y -> sampled y p)
                 (Cons p1 (Cons p2 (Cons p3 Nil)))
                 (Cons d1 (Cons d2 (Cons d3 Nil))) /\
               ((World !st interp) |= Possible (t_m1 $< t_m2)) /\
               ((World !st interp) |= Possible (t_m1 $> t_m2)) /\
               ((World !st interp) |= Possible (t_m1 $< t_m3)) /\
               ((World !st interp) |= Possible (t_m1 $> t_m3)) /\
               ((World !st interp) |= Possible (t_m2 $< t_m3)) /\
               ((World !st interp) |= Possible (t_m2 $> t_m3))
             }
    ensures {
      let p = result in
      let h = (t_m1 $!= t_m2) $|| (t_m1 $!= t_m3) $|| (t_m2 $!= t_m3) in
      Eq p = compose_pvs h !st &&
      (World !st interp) |= StatB (Eq p) h
    }
    = exec_kruskal (Cons p1 (Cons p2 (Cons p3 Nil))) (Cons d1 (Cons d2 (Cons d3
        Nil)))
end
```

You can verify the code above by running:

```
$ statwhy examples/mlw/ex_kruskal.mlw
```


Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Time
Task ex_kruskal.mlw
Messages Log Edited proof Prover output Counterexample

```
ex_kruskal.mlw
  Kruskal_example
    ex_kruskal'vc [VC for ex_kruskal]
      subst_all_statwhy
        0 [VC for ex_kruskal]
          compute_specified_statwhy
            0 [VC for ex_kruskal]
              CVC5 1.3.1                    1.00 (steps: 1
              split_vc
                0 [precondition]
                1 [postcondition]
                  CVC5 1.3.1                1.00 (steps: 1
                  split_vc
                    0 [postcondition]
                    1 [postcondition]
```

```

  let ex_kruskal (d1 d2 d3 : dataset real)
  (* Executes Kruskal test for 3 population medians *)
    requires { for_all (fun d -> d.scale = Interval) (Cons d1 (
               independent_list (Cons d1 (Cons d2 (Cons d3 Nil)
               is_empty !st /\
               for_all2
                 (fun p y -> sampled y p)
                 (Cons p1 (Cons p2 (Cons p3 Nil)))
                 (Cons d1 (Cons d2 (Cons d3 Nil))) /\
               ((World !st interp) |= Possible (t_m1 $< t_m2))
               ((World !st interp) |= Possible (t_m1 $> t_m2))
               ((World !st interp) |= Possible (t_m1 $< t_m3))
               ((World !st interp) |= Possible (t_m1 $> t_m3))
               ((World !st interp) |= Possible (t_m2 $< t_m3))
               ((World !st interp) |= Possible (t_m2 $> t_m3))
             }
    ensures {
      let p = result in
      let h = (t_m1 $!= t_m2) $|| (t_m1 $!= t_m3) $|| (t_m2 $!=
      Eq p = compose_pvs h !st &&
      (World !st interp) |= StatB (Eq p) h
```

```
0/0/0 type commands here
```

```
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
The transformation made no progress
```

6.5 Tukey's HSD Test

In this section, we show the verification of a program for a popular multiple comparison test called the *Tukey's HSD test*.

Given datasets from multiple groups, Tukey's HSD test examines the differences in means for each pair of groups. In StatWhy, the specification of the test is as follows:

```
  val exec_tukey_hsd (dists : list distribution) (ys : list (dataset real)) :
      array real
  writes { st }
  requires {
    independent_list ys /\
    Length.length dists = Length.length ys /\
    for_all (fun p -> (World !st interp) |= is_normal p) dists /\
    for_all2 (fun p y -> sampled y p) dists ys /\
    for_all (fun y -> scale_leq Interval y.scale) ys /\
    for_all
      (fun t -> let (x, y) = t in
        (World !st interp) |= eq_variance x y)
      (cmb dists) /\
    for_all
      (fun t -> let (s, t) = t in
        ((World !st interp) |= Possible (mean s $< mean t) /\
         (World !st interp) |= Possible (mean s $> mean t)))
      (cmb_term dists)
  }
  ensures {
    let ps = result in
    length ps = div2 ((Length.length dists) * (Length.length dists - 1)) /\
    let b = length ps in
    (forall i : int. 0 <= i < b -> pvalue (ps[i])) /\
    let cmbs = combinations (map (fun p -> RealT (mean p)) dists) "!=" in
    !st = (cmb_store cmbs ps 0) ++ !(old st)
  }
end
```

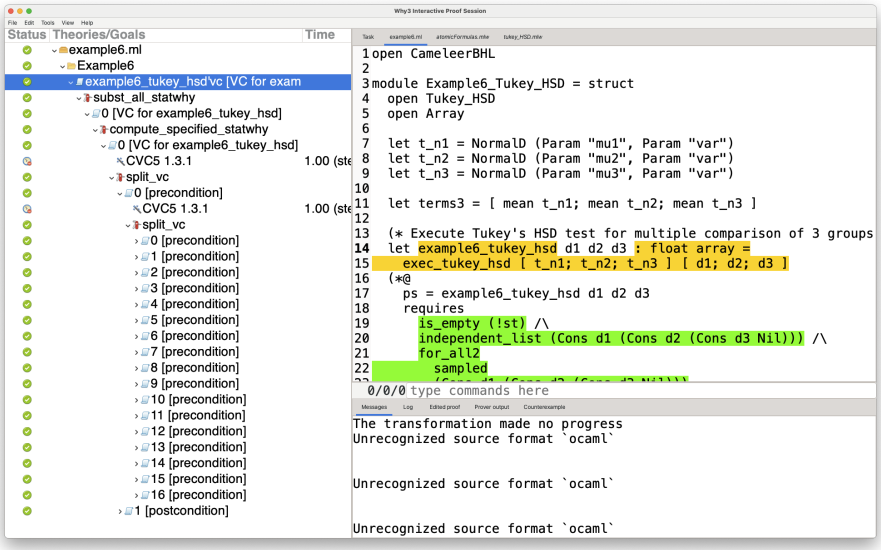
`dists` denotes a list of population distributions, and `ys` is a list of the datasets sampled from the distributions. `exec_tukey_hsd` returns an array of p-values. These p-values are sorted in lexicographic order, such as $p_{12}, p_{13}, p_{14}, p_{23}, p_{24}, p_{34}$, where p_{ij} is the p-value for the comparison of groups i and j.

`exec_tukey_hsd` requires that all populations have the same variance, which is specified by

```
  for_all
      (fun t -> let (x, y) = t in
        (World !st interp) |= eq_variance x y)
      (combinations_poly ppls)
```

`example6_tukey_hsd` in `examples/example6.ml` performs Tukey's HSD test for three groups. To verify the code, execute the following command:

```
$ statwhy examples/example6.ml
```


Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Time
Task example6.ml atomicFormulas.mlw tukey_HSD.mlw
Messages Log Edited proof Prover output Counterexample

```
example6.ml
  Example6
    example6_tukey_hsd'vc [VC for exam
      subst_all_statwhy
        0 [VC for example6_tukey_hsd]
          compute_specified_statwhy
            0 [VC for example6_tukey_hsd]
              CVC5 1.3.1                        1.00 (ste
              split_vc
                0 [precondition]
                  CVC5 1.3.1                    1.00 (ste
                  split_vc
                    0 [precondition]
                    1 [precondition]
                    2 [precondition]
                    3 [precondition]
                    4 [precondition]
                    5 [precondition]
                    6 [precondition]
                    7 [precondition]
                    8 [precondition]
                    9 [precondition]
                    10 [precondition]
                    11 [precondition]
                    12 [precondition]
                    13 [precondition]
                    14 [precondition]
                    15 [precondition]
                    16 [precondition]
                1 [postcondition]
```

```
open CameleerBHL

module Example6_Tukey_HSD = struct
  open Tukey_HSD
  open Array

  let t_n1 = NormalD (Param "mu1", Param "var")
  let t_n2 = NormalD (Param "mu2", Param "var")
  let t_n3 = NormalD (Param "mu3", Param "var")

  let terms3 = [ mean t_n1; mean t_n2; mean t_n3 ]

  (* Execute Tukey's HSD test for multiple comparison of 3 groups
  let example6_tukey_hsd d1 d2 d3 : float array =
    exec_tukey_hsd [ t_n1; t_n2; t_n3 ] [ d1; d2; d3 ]
  (*@
    ps = example6_tukey_hsd d1 d2 d3
    requires
      is_empty (!st) /\
      independent_list (Cons d1 (Cons d2 (Cons d3 Nil))) /\
      for_all2
        sampled
```

```
0/0/0 type commands here
```

```
The transformation made no progress
Unrecognized source format `ocaml`


Unrecognized source format `ocaml`


Unrecognized source format `ocaml`
```

7 StatWhy's Scope and Scalability

7.1 List of Hypothesis Testing Methods Supported in StatWhy

We show the list of all the hypothesis testing methods supported in StatWhy in Table 1. Most of the supported tests are implemented based on the specification of hypothesis testing functions in `scipy.stats`. In future work, we will support the hypothesis testing methods that have not supported in this version of StatWhy (e.g., those for correlation).

7.2 Scalability of StatWhy

We conducted experiments to evaluate the performance of the program verification using StatWhy. We performed the experiments on a MacBook Pro with Apple M2 Max CPU and 96 GB memory using the external SMT solver cvc5 1.0.6. We presented part of the following experimental results in our paper [3].

Table 2 summarizes the execution times for StatWhy to verify programs for popular single-comparison hypothesis testing methods. The verification of these testing methods is very efficient even when precondition formulas are relatively large (e.g., in Alexander-Govern test and in χ^2 test).

Table 3 shows the execution times for StatWhy to verify hypothesis testing programs for practical numbers of disjunctive/conjunctive hypotheses. These experiments took roughly the same amount of time for a larger number of hypotheses.

Table 4 presents the execution times (sec) for the most common multiple comparison methods described in standard textbooks. The numbers of groups compared in the experiments are practical but challenging, as the number of comparisons grows rapidly with the number of groups. The verification of these programs is efficient, since our proof strategy explained in [3] accelerates the proof search for programs with folding operations and test histories.

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., Kobayashi, K., Suenaga, K.: Statwhy: Formal verification tool for statistical hypothesis testing programs. In: Proc. the 37th International Conference on Computer Aided Verification (CAV 2025), Part II. Lecture Notes in Computer Science, vol. 15932, pp. 216–230. Springer (2025). https://doi.org/10.1007/978-3-031-98679-6_10
4. Kawamoto, Y., Sato, T., Suenaga, K.: Formalizing statistical beliefs in hypothesis testing using program logic. In: Proc. the 18th International Conference on Principles of Knowledge Representation and Reasoning (KR'21). pp. 411–421 (2021). https://doi.org/10.24963/kr.2021/39
5. 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
6. 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

Names of testing methods	Goals of the hypothesis testing	File names	Names in `scipy.stats`
t-test	mean of a population	ttest.mlw	ttest_1samp
	two population means (independent samples)	ttest.mlw	ttest_ind
	two population means (paired comparisons)	ttest.mlw	ttest_rel
χ^2-test	goodness of fit	chisquare.mlw	chisquare
	independence of two categorical data	chi2_contingency.mlw	chi2_contingency
F-test	two population variances	ftest.mlw	
Binomial test	probability of success	binom.mlw	binomtest
Quantile test	quantile of a population	quantile_test.mlw	quantile_test
Skewness test	skewness of a population	skew_test.mlw	skew_test
D'Agostino-Pearson's test	normality of a population	normality_test.mlw	normaltest
Jarque-Bera test	normality of a population	normality_test.mlw	jarque_bera
Shapiro-Wilk test	normality of a population	normality_test.mlw	shapiro
Anderson-Darling test	population type	anderson.mlw	anderson
Cramér-von Mises test	goodness of fit	cramervonmises.mlw	cramervonmises
Power divergence test	goodness of fit	power_divergence.mlw	power_divergence
Wilcoxon signed-rank test	equality of two distributions	wilcoxon.mlw	wilcoxon
Mann-Whitney U test	equality of two distributions	mannwhitneyu.mlw	mannwhitneyu
Baumgartner-Weiss-Schindler test	equality of two distributions	bws_test.mlw	bws_test
Wilcoxon rank-sum test	equality of two distributions	ranksums.mlw	ranksums
Two-sample Cramér-von Mises test	equality of two distributions	cramervonmises.mlw	cramervonmises_2samp
Epps-Singleton test	equality of two distributions	epps_singleton_2samp.mlw	epps_singleton_2samp
Two-sample Kolmogorov-Smirnov test	equality of two distributions	ks.ml	ks_2samp
Mood's median test	two population medians	median_test.mlw	median_test
Fisher's exact test	independence of two categorical data	fisher_exact.mlw	fisher_exact
One-way ANOVA	equality of population means	oneway_ANOVA.mlw	f_oneway
Tukey's HSD test	pairwise comparison of means	tukey_HSD.mlw	tukey_hsd
Steel-Dwass test	pairwise comparison of means	steel_dwass.mlw	
Dunnett's test	comparisons of means against a control group	dunnett.mlw	dunnett
Williams' test	comparisons of means against a control group	williams.mlw	
Steel test	comparisons of means against a control group	steel.mlw	
Kruskal-Wallis H-test	equality of medians	kruskal.mlw	kruskal
Alexander-Govern test	equality of means	alexandergovern.mlw	alexandergovern
Fligner-Killeen test	equality of variances	fligner.mlw	fligner
Levene test	equality of variances	levene.mlw	levene
Bartlett's test	equality of variances	bartlett.mlw	bartlett
Friedman test	consistency among samples	friedmanchisquare.mlw	friedmanchisquare
k-sample Anderson-Darling test	equality of populations	anderson.mlw	anderson_ksamp

Table 1: Hypothesis testing methods supported in StatWhy

Table 2: The execution times (sec) of programs for popular single-comparison hypothesis testing.

File names	Test methods	Times (sec)
ex_alexandergovern.mlw	Alexander-Govern test	8.75
ex_bartlett.mlw	Bartlett's test	0.87
ex_binom.mlw	Binomial test	0.46
ex_chisquare.mlw	χ^2 test	5.50
ex_ftest.mlw	F-test	0.45
ex_kruskal.mlw	Kruskal-Wallis test	0.88
ex_oneway.mlw	One-way ANOVA	2.55
ex_ttest.mlw	One-sample t-test	0.46
	Student's t-test	0.48
	Welch's t-test	0.43
	Paired t-test	0.44

Table 3: The execution times (sec) for verifying hypothesis testing programs with different numbers of disjunctive (OR) and conjunctive (AND) hypotheses. In practice, the number of hypotheses in hypothesis testing is less than 10.

#hypotheses	2	3	4	5	6	7	8	9	10
OR	8.77	8.89	8.84	8.94	9.01	9.01	9.04	9.16	9.23
AND	8.82	8.72	8.86	8.98	8.95	9.03	9.11	9.17	9.46

Table 4: The execution times (sec) for various multiple comparison methods. #groups (resp. #comparisons) represents the number of groups (resp. combinations of groups) compared in the hypothesis testing. In practice, #groups in multiple comparison is at most 7.

		#groups					
Test methods	Metric	2	3	4	5	6	7
Tukey's HSD test	Times (sec)	0.37	9.09	9.33	9.81	15.27	16.39
	#comparisons	1	3	6	10	15	21
Dunnett's test	Times (sec)	0.48	8.98	9.17	9.61	9.62	9.77
	#comparisons	1	2	3	4	5	6
Williams' test	Times (sec)	0.48	8.90	9.04	9.16	9.23	9.58
	#comparisons	1	2	3	4	5	6
Steel-Dwass' test	Times (sec)	0.44	9.05	9.43	9.76	15.10	16.24
	#comparisons	1	3	6	10	15	21
Steel's test	Times (sec)	0.49	8.79	8.92	9.11	9.43	9.74
	#comparisons	1	2	3	4	5	6