

# User Documentation for StatWhy v.1.3.0

Yusuke Kawamoto<sup>1</sup>, Kentaro Kobayashi<sup>1,2</sup>, and Kohei Suenaga<sup>3</sup> \*

<sup>1</sup> National Institute of Advanced Industrial Science and Technology (AIST), Tokyo,  
Japan

<sup>2</sup> University of Tsukuba, Ibaraki, Japan  
<sup>3</sup> 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 JP-MJPR2022, 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:<sup>4</sup>

```
$ 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.<sup>5</sup>

**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)

<sup>4</sup> Using the versions later than OCaml 5.0, we encountered a dynamic link issue with Cameleer on macOS.

<sup>5</sup> 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.

## 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`.<sup>6</sup> 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.

- `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.

---

<sup>6</sup> Programmers are required to use the WhyML language to describe specifications.

- `d.scale = Interval`<sup>7</sup> 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_1` and `(World (!st) interp) |= Possible fmlA_u` represent that the analyst has a prior belief that the *lower-tail* hypothesis  $fmlA_1 \stackrel{\text{def}}{=} (\text{mean}(x) < 1.0)$  may be true, and that the *upper-tail* hypothesis  $fmlA_u \stackrel{\text{def}}{=} (\text{mean}(x) > 1.0)$  may be true<sup>8</sup>. 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  $A \rightarrow B$ .

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.
- 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.

---

<sup>7</sup> `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.

<sup>8</sup> We remark that the disjunction `fmlA_1 ∨ fmlA_u` is logically equivalent to `fmlA`.

## 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_1)`, 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.,

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

where  $y \sim N(\mu, \sigma^2)$  represents that a dataset  $y$  is sampled from the population that follows a normal distribution  $N(\mu, \sigma^2)$  with an unknown mean  $\mu$  and an unknown variance  $\sigma^2$ . The modal formula  $\mathbf{P}\varphi$  represents that, before conducting the hypothesis test, we have the *prior belief* that the alternative hypothesis  $\varphi$  *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.,

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

Intuitively, by a hypothesis test  $A$  on the dataset  $y$ , we believe  $\varphi$  with a  $p$ -value  $\alpha \leq 0.05$ . Since the result of the hypothesis test may be wrong, we use the belief modality  $\mathbf{K}_{y,A}^{\leq 0.05}$  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*:

$$\Gamma \vdash \{\psi_{\text{pre}}\} C \{\psi^{\text{post}}\}, \quad (2)$$

representing that whenever the precondition  $\psi_{\text{pre}}$  is satisfied, the execution of the program  $C$  results in the satisfaction of the postcondition  $\psi^{\text{post}}$ . 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:<sup>9</sup>

```
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.<sup>10</sup>

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:

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

<sup>9</sup> The actual definition can be found in `logicalFormula.mlw`.

<sup>10</sup> These folding operations are defined in `hof.mlw` and `utility.mlw`.

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.<sup>11</sup>

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  $(\text{cmb } \text{dists})$  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.

## 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.

---

<sup>11</sup> 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.

- 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.

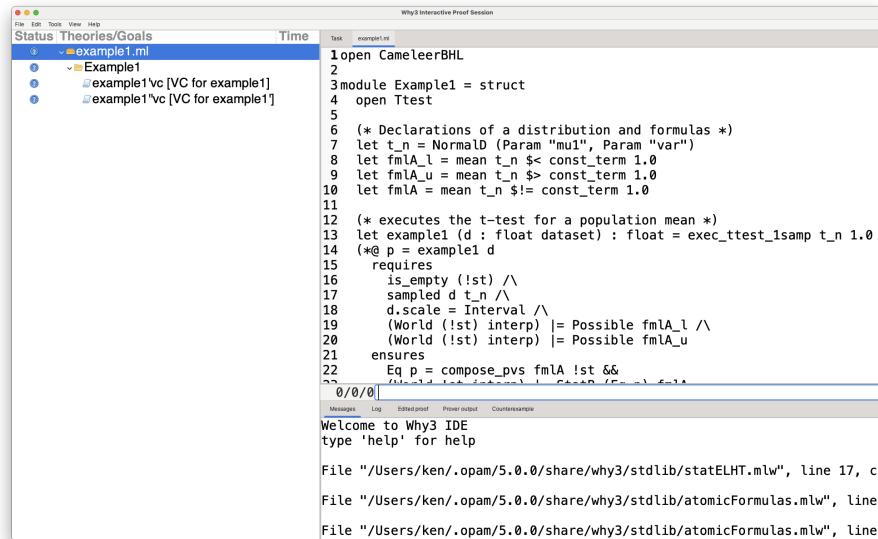
## 6.1 Simple *t*-test (One-Sample *t*-test)

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.



The screenshot shows the Why3 IDE interface. The left panel displays the 'Status Theories/Goals' window with the file 'example1.ml' open, showing two generated verification conditions: 'Example1' and 'example1'vc [VC for example1]. The right panel shows the 'Task' window with the following OCaml code:

```

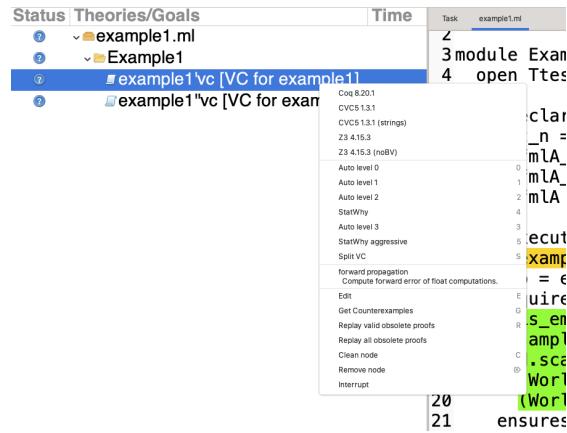
open CameleerBHL
module Example1 = struct
  open Ttest
  (* Declarations of a distribution and formulas *)
  let t_n = NormalD (Param "mu1", Param "var")
  let fmlA_l = mean t_n $< const_term 1.0
  let fmlA_u = mean t_n $> const_term 1.0
  let fmlA = mean t_n $!= const_term 1.0
  (* executes the t-test 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 &&
    (World (!st) interp) |= Possible fmlA
  0/0/0
  
```

The status bar at the bottom indicates '0/0/0'. The bottom of the screen shows the 'Welcome to Why3 IDE' message and the file paths for the loaded files: '/Users/ken/.opam/5.0.0/share/why3/stdlib/statELHT.mlw', '/Users/ken/.opam/5.0.0/share/why3/stdlib/atomicFormulas.mlw', and '/Users/ken/.opam/5.0.0/share/why3/stdlib/atomicFormulas.mlw'.

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: `sampling 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.



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:

```
Why3 Interactive Proof Session
Status Theories/Goals
  1 example1.ml
  2 Example1
  3 example1`vc [VC for example1]
  4 subst_all_statwhy
  5 compute_specified_statwhy
  6 #0 [VC for example1]
  7 #0 [VC for example1]
  8 example1`vc [VC for example1]

Task example1.ml
1
2
3 module Example1 = struct
4   open Ttest
5
6 (* Declarations of a distribution and formulas *)
7 let t_n = NormalD (Param "mu1", Param "var")
8 let fmlA_l = mean t_n $< const_term 1.0
9 let fmlA_u = mean t_n $> const_term 1.0
10 let fmlA = mean t_n $!= const_term 1.0
11
12 (* executes the t-test for a population mean *)
13 let example1 (d : float dataset) : float = exec_ttest_isamp t_n 1.0
14 (*@ p = example1 d
15   requires
16   is_empty (!st) /\ 
17   sampled d t_n /\ 
18   d.scale = Interval /\ 
19   (World (!st) interp) |= Possible fmlA_l /\ 
20   (World (!st) interp) |= Possible fmlA_u
21   ensures
22   Eq p = compose_pvs fmlA !st &&
23   (World !st interp) |= StatB (Eq p) fmlA
24 *)
0/0 type commands here
The transformation made no progress
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`.

The screenshot shows the Why3 Interactive Proof Session window. On the left, there is a tree view of the proof state, labeled "Status Theories/Goals". It shows a main node "example1.ml" which branches into "Example1", "example1'vc [VC for example1]", "subst\_all\_statwhy", "compute\_specified\_statwhy", and "split\_vc". "split\_vc" further branches into "CVC5 1.3.1", "split\_vc", and "0 [precondition]". "0 [precondition]" branches into "CVC5 1.3.1", "split\_vc", and "0 [precondition]". "0 [precondition]" branches into "CVC5 1.3.1", "0 [precondition]", and "1 [postcondition]". On the right, there is a text-based interface for entering commands. The current command being entered is "example1'vc :". Below the input field, the message "The transformation made no progress" is displayed. At the bottom, there are tabs for "Messages", "Log", "Edited proof", "Pretty output", and "Counterexample".

**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  $y_1$  and  $y_2$  must be sampled independently (not in pair).
- `(World !st interp) |= eq_variance d1 d2` specifies that the population distributions  $d_1$  and  $d_2$  have the same variance.
- `(World !st interp) |= Not (check_variance d1)` and `(World !st interp) |= Not (check_variance d2)` specify that the variances of  $d_1$  and  $d_2$  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.

```

File Edit Tools View Help
Status Theories/Goals
example2.ml
  Example2
    example2vc [VC for example2]
      subst_all_statwhy
        0 [VC for example2]
          compute_specified_statwhy
            #0 [VC for example2]

Task example2.ml
3module Example2 = struct
4  open Ttest
5
6  let t_n1 = NormalD (Param "mu1", Param "var1")
7  let t_n2 = NormalD (Param "mu2", Param "var2")
8  let fmlA_l : formula = mean t_n1 $< mean t_n2
9  let fmlA_u : formula = mean t_n1 $> mean t_n2
10 let fmlA : formula = mean t_n1 $!= mean t_n2
11
12 (* Execute Paired t-test for two population means *)
13 let example2 (d1: float dataset) (d2: float dataset) : float =
14  exec_ttest_paired t_n1 t_n2 d1 d2 Two
15 (*@
16  p = example2 d1 d2
17  requires
18  is_empty (!st) /\ paired d1 d2 /\ 
19  d1.scale = Interval /\ d2.scale = Interval /\ 
20  sampled d1 t_n1 /\ sampled d2 t_n2 /\ 
21  (World (list) interp) |= Possible fmlA_l /\ 
22  (World (list) interp) |= Possible fmlA_u
23  ensures
24  Ens n = compose nvs fmlA !st &&
0/0/0 type commands here

```

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)

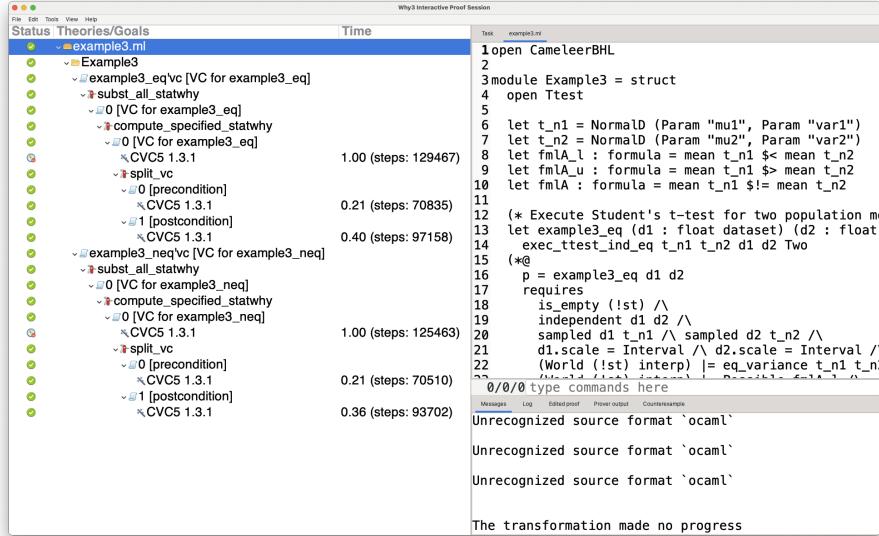
```

```

| 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`



### 6.3 Dealing with Combined Tests in StatWhy

Let  $A_{\varphi_0}$  and  $A_{\varphi_1}$  be two hypothesis tests with alternative hypotheses  $\varphi_0$  and  $\varphi_1$ , respectively. There are two possible combinations of  $A_{\varphi_0}$  and  $A_{\varphi_1}$ . 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  $A_{\varphi_0}$  and  $A_{\varphi_1}$  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  $A_{\varphi_0}$  and  $A_{\varphi_1}$  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 \wedge fmlB) \vee 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`.

```

File Edit Tools View Help
Status Theories/Goals
  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 [precondition]
                              1 [postcondition]
                                1 [postcondition]
                                  1 [postcondition]
                                    example_and_orvc [VC for example_and_or]
                                    example_or_andvc [VC for example_or_and]
                                    example_and_andvc [VC for example_and_and]

```

```

Task example4.ml
Time 1.00 (s)

21 let fmlA_and_and = fmlA $& fmlB $& fmlC
22 (* H1 : (fmlA \vee fmlB) \vee fmlC *)
23 let example_or_or d1 d2 d3 : float =
24   let p1 = exec_ttest_ind_eq t_n1 t_n2 d1 d2 Two in
25   let p2 = exec_ttest_ind_eq t_n1 t_n3 d1 d3 Two in
26   let p3 = exec_ttest_ind_eq t_n2 t_n3 d2 d3 Two in
27   p1 +. p2 +. p3
28
29 (*@
30   p = example_or_or d1 d2 d3
31   requires
32     ls_empty (!st) /\
33     sampled d1 t_n1 /\ sampled d2 t_n2 /\ sampled d3 t_n3 /\
34     d1.scale = d2.scale = d3.scale = Interval /\
35     independent d1 d2 /\ independent d1 d3 /\ independent d2 d3 /\
36     (World (!st) interp) |= Possible fmlA_l /\
37     (World (!st) interp) |= Possible fmlA_u /\
38     (World (!st) interp) |= Possible fmlB_l /\
39     (World (!st) interp) |= Possible fmlB_u /\
40     (World (!st) interp) |= Possible fmlC_l /\
41     (World (!st) interp) |= Possible fmlC_u
42   ensures
43 
```

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  $\varphi_0$  and  $\varphi_1$ . 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`.

**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 (Lseq (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

```

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

The screenshot shows the Why3 Interactive Proof Session interface. The left pane displays the theory and goals, while the right pane shows the OCaml code and its verification status. The code defines a module `Example5` with a struct containing fields `t_n`, `fmlA_l`, `fmlA_u`, and `fmlA`. It also includes functions for `exec_ttest_isamp` and `exec_tttest_isamp`, and a `min` function. The verification status indicates that the program is INCORRECT and fails verification. The right pane also shows error messages related to type commands and source format.

```

Status: Theories/Goals
File Edit Tools View Help
Status: Theories/Goals
Time: Task: example5.ml
3 module Example5 = struct
4   open Ttest
5
6   let t_n = NormalD (Param "mu1", Param "var")
7   let fmlA_l = mean t_n $< const_term 1.0
8   let fmlA_u = mean t_n $> const_term 1.0
9   let fmlA = mean t_n $!= const_term 1.0
10
11  (* Example of p-value hacking *)
12  (* This program is INCORRECT and so its verification FAIL
13  let examples d1 d2 =
14    let p1 = exec_ttest_isamp t_n 1.0 d1 Two in
15    let p2 = exec_tttest_isamp t_n 1.0 d2 Two in
16    let p = min p1 p2 in
17    (p1, p2, p)
18  (*@
19  (p1, p2, p) = ex_hack d1 d2
20  requires
21  is_empty (!st) /\
22  sampled d1 t_n /\ sampled d2 t_n /\
23  d1.scale = d2.scale = Interval /\ 
24  (World (!st) interp) |= Possible fmlA_l /\ 
25  (World (!st) interp) |= Possible fmlA_u
26
27  #0 [postcondition]
28
29  #1 [postcondition]
30  #2 [postcondition]
31  #3 [postcondition]
32
33  #CVC5 1.3.1
34
35  #split_vc
36  #0 [precondition]
37  #1 [precondition]
38  #2 [postcondition]
39  #CVC5 1.3.1
40
41  #split_vc
42  #0 [postcondition]
43  #compute_in_goal_statwhy
44  #1 [postcondition]
45  #0 [postcondition]
46  #0 [postcondition]
47  #compute_in_goal_statwhy
48
49  #CVC5 1.3.1
50
51  #0 [postcondition]
52  #compute_in_goal_statwhy
53  #1 [postcondition]
54  #0 [postcondition]
55  #0 [postcondition]
56  #compute_in_goal_statwhy
57
58  #CVC5 1.3.1
59
60  #0 [postcondition]
61
62  #1 [postcondition]
63  #2 [postcondition]
64  #3 [postcondition]
65
66  #CVC5 1.3.1
67
68  #0 [postcondition]
69
70  #1 [postcondition]
71  #2 [postcondition]
72  #3 [postcondition]
73
74  #CVC5 1.3.1
75
76  #0 [postcondition]
77
78  #1 [postcondition]
79  #2 [postcondition]
80  #3 [postcondition]
81
82  #CVC5 1.3.1
83
84  #0 [postcondition]
85
86  #1 [postcondition]
87  #2 [postcondition]
88  #3 [postcondition]
89
90  #CVC5 1.3.1
91
92  #0 [postcondition]
93
94  #1 [postcondition]
95  #2 [postcondition]
96  #3 [postcondition]
97
98  #CVC5 1.3.1
99
100 #0 [postcondition]
101
102 #1 [postcondition]
103 103 #2 [postcondition]
104 104 #3 [postcondition]
105
106 106 #CVC5 1.3.1
107
108 #0 [postcondition]
109
110 #1 [postcondition]
111 111 #2 [postcondition]
112 112 #3 [postcondition]
113
114 114 #CVC5 1.3.1
115
116 #0 [postcondition]
117
118 #1 [postcondition]
119 119 #2 [postcondition]
120 120 #3 [postcondition]
121
122 122 #CVC5 1.3.1
123
124 #0 [postcondition]
125
126 #1 [postcondition]
127 127 #2 [postcondition]
128 128 #3 [postcondition]
129
130 130 #CVC5 1.3.1
131
132 #0 [postcondition]
133
134 #1 [postcondition]
135 135 #2 [postcondition]
136 136 #3 [postcondition]
137
138 138 #CVC5 1.3.1
139
140 #0 [postcondition]
141
142 #1 [postcondition]
143 143 #2 [postcondition]
144 144 #3 [postcondition]
145
146 146 #CVC5 1.3.1
147
148 #0 [postcondition]
149
150 #1 [postcondition]
151 151 #2 [postcondition]
152 152 #3 [postcondition]
153
154 154 #CVC5 1.3.1
155
156 #0 [postcondition]
157
158 #1 [postcondition]
159 159 #2 [postcondition]
160 160 #3 [postcondition]
161
162 162 #CVC5 1.3.1
163
164 #0 [postcondition]
165
166 #1 [postcondition]
167 167 #2 [postcondition]
168 168 #3 [postcondition]
169
170 170 #CVC5 1.3.1
171
172 #0 [postcondition]
173
174 #1 [postcondition]
175 175 #2 [postcondition]
176 176 #3 [postcondition]
177
178 178 #CVC5 1.3.1
179
180 #0 [postcondition]
181
182 #1 [postcondition]
183 183 #2 [postcondition]
184 184 #3 [postcondition]
185
186 186 #CVC5 1.3.1
187
188 #0 [postcondition]
189
190 #1 [postcondition]
191 191 #2 [postcondition]
192 192 #3 [postcondition]
193
194 194 #CVC5 1.3.1
195
196 #0 [postcondition]
197
198 #1 [postcondition]
199 199 #2 [postcondition]
200 200 #3 [postcondition]
201
202 202 #CVC5 1.3.1
203
204 #0 [postcondition]
205
206 #1 [postcondition]
207 207 #2 [postcondition]
208 208 #3 [postcondition]
209
210 210 #CVC5 1.3.1
211
212 #0 [postcondition]
213
214 #1 [postcondition]
215 215 #2 [postcondition]
216 216 #3 [postcondition]
217
218 218 #CVC5 1.3.1
219
220 #0 [postcondition]
221
222 #1 [postcondition]
223 223 #2 [postcondition]
224 224 #3 [postcondition]
225
226 226 #CVC5 1.3.1
227
228 #0 [postcondition]
229
230 #1 [postcondition]
231 231 #2 [postcondition]
232 232 #3 [postcondition]
233
234 234 #CVC5 1.3.1
235
236 #0 [postcondition]
237
238 #1 [postcondition]
239 239 #2 [postcondition]
240 240 #3 [postcondition]
241
242 242 #CVC5 1.3.1
243
244 #0 [postcondition]
245
246 #1 [postcondition]
247 247 #2 [postcondition]
248 248 #3 [postcondition]
249
250 250 #CVC5 1.3.1
251
252 #0 [postcondition]
253
254 #1 [postcondition]
255 255 #2 [postcondition]
256 256 #3 [postcondition]
257
258 258 #CVC5 1.3.1
259
260 #0 [postcondition]
261
262 #1 [postcondition]
263 263 #2 [postcondition]
264 264 #3 [postcondition]
265
266 266 #CVC5 1.3.1
267
268 #0 [postcondition]
269
270 #1 [postcondition]
271 271 #2 [postcondition]
272 272 #3 [postcondition]
273
274 274 #CVC5 1.3.1
275
276 #0 [postcondition]
277
278 #1 [postcondition]
279 279 #2 [postcondition]
280 280 #3 [postcondition]
281
282 282 #CVC5 1.3.1
283
284 #0 [postcondition]
285
286 #1 [postcondition]
287 287 #2 [postcondition]
288 288 #3 [postcondition]
289
290 290 #CVC5 1.3.1
291
292 #0 [postcondition]
293
294 #1 [postcondition]
295 295 #2 [postcondition]
296 296 #3 [postcondition]
297
298 298 #CVC5 1.3.1
299
300 #0 [postcondition]
301
302 #1 [postcondition]
303 303 #2 [postcondition]
304 304 #3 [postcondition]
305
306 306 #CVC5 1.3.1
307
308 #0 [postcondition]
309
310 #1 [postcondition]
311 311 #2 [postcondition]
312 312 #3 [postcondition]
313
314 314 #CVC5 1.3.1
315
316 #0 [postcondition]
317
318 #1 [postcondition]
319 319 #2 [postcondition]
320 320 #3 [postcondition]
321
322 322 #CVC5 1.3.1
323
324 #0 [postcondition]
325
326 #1 [postcondition]
327 327 #2 [postcondition]
328 328 #3 [postcondition]
329
330 330 #CVC5 1.3.1
331
332 #0 [postcondition]
333
334 #1 [postcondition]
335 335 #2 [postcondition]
336 336 #3 [postcondition]
337
338 338 #CVC5 1.3.1
339
340 #0 [postcondition]
341
342 #1 [postcondition]
343 343 #2 [postcondition]
344 344 #3 [postcondition]
345
346 346 #CVC5 1.3.1
347
348 #0 [postcondition]
349
350 #1 [postcondition]
351 351 #2 [postcondition]
352 352 #3 [postcondition]
353
354 354 #CVC5 1.3.1
355
356 #0 [postcondition]
357
358 #1 [postcondition]
359 359 #2 [postcondition]
360 360 #3 [postcondition]
361
362 362 #CVC5 1.3.1
363
364 #0 [postcondition]
365
366 #1 [postcondition]
367 367 #2 [postcondition]
368 368 #3 [postcondition]
369
370 370 #CVC5 1.3.1
371
372 #0 [postcondition]
373
374 #1 [postcondition]
375 375 #2 [postcondition]
376 376 #3 [postcondition]
377
378 378 #CVC5 1.3.1
379
380 #0 [postcondition]
381
382 #1 [postcondition]
383 383 #2 [postcondition]
384 384 #3 [postcondition]
385
386 386 #CVC5 1.3.1
387
388 #0 [postcondition]
389
390 #1 [postcondition]
391 391 #2 [postcondition]
392 392 #3 [postcondition]
393
394 394 #CVC5 1.3.1
395
396 #0 [postcondition]
397
398 #1 [postcondition]
399 399 #2 [postcondition]
400 400 #3 [postcondition]
401
402 402 #CVC5 1.3.1
403
404 #0 [postcondition]
405
406 #1 [postcondition]
407 407 #2 [postcondition]
408 408 #3 [postcondition]
409
410 410 #CVC5 1.3.1
411
412 #0 [postcondition]
413
414 #1 [postcondition]
415 415 #2 [postcondition]
416 416 #3 [postcondition]
417
418 418 #CVC5 1.3.1
419
420 #0 [postcondition]
421
422 #1 [postcondition]
423 423 #2 [postcondition]
424 424 #3 [postcondition]
425
426 426 #CVC5 1.3.1
427
428 #0 [postcondition]
429
430 #1 [postcondition]
431 431 #2 [postcondition]
432 432 #3 [postcondition]
433
434 434 #CVC5 1.3.1
435
436 #0 [postcondition]
437
438 #1 [postcondition]
439 439 #2 [postcondition]
440 440 #3 [postcondition]
441
442 442 #CVC5 1.3.1
443
444 #0 [postcondition]
445
446 #1 [postcondition]
447 447 #2 [postcondition]
448 448 #3 [postcondition]
449
450 450 #CVC5 1.3.1
451
452 #0 [postcondition]
453
454 #1 [postcondition]
455 455 #2 [postcondition]
456 456 #3 [postcondition]
457
458 458 #CVC5 1.3.1
459
460 #0 [postcondition]
461
462 #1 [postcondition]
463 463 #2 [postcondition]
464 464 #3 [postcondition]
465
466 466 #CVC5 1.3.1
467
468 #0 [postcondition]
469
470 #1 [postcondition]
471 471 #2 [postcondition]
472 472 #3 [postcondition]
473
474 474 #CVC5 1.3.1
475
476 #0 [postcondition]
477
478 #1 [postcondition]
479 479 #2 [postcondition]
480 480 #3 [postcondition]
481
482 482 #CVC5 1.3.1
483
484 #0 [postcondition]
485
486 #1 [postcondition]
487 487 #2 [postcondition]
488 488 #3 [postcondition]
489
490 490 #CVC5 1.3.1
491
492 #0 [postcondition]
493
494 #1 [postcondition]
495 495 #2 [postcondition]
496 496 #3 [postcondition]
497
498 498 #CVC5 1.3.1
499
500 #0 [postcondition]
501
502 #1 [postcondition]
503 503 #2 [postcondition]
504 504 #3 [postcondition]
505
506 506 #CVC5 1.3.1
507
508 #0 [postcondition]
509
510 #1 [postcondition]
511 511 #2 [postcondition]
512 512 #3 [postcondition]
513
514 514 #CVC5 1.3.1
515
516 #0 [postcondition]
517
518 #1 [postcondition]
519 519 #2 [postcondition]
520 520 #3 [postcondition]
521
522 522 #CVC5 1.3.1
523
524 #0 [postcondition]
525
526 #1 [postcondition]
527 527 #2 [postcondition]
528 528 #3 [postcondition]
529
530 530 #CVC5 1.3.1
531
532 #0 [postcondition]
533
534 #1 [postcondition]
535 535 #2 [postcondition]
536 536 #3 [postcondition]
537
538 538 #CVC5 1.3.1
539
540 #0 [postcondition]
541
542 #1 [postcondition]
543 543 #2 [postcondition]
544 544 #3 [postcondition]
545
546 546 #CVC5 1.3.1
547
548 #0 [postcondition]
549
550 #1 [postcondition]
551 551 #2 [postcondition]
552 552 #3 [postcondition]
553
554 554 #CVC5 1.3.1
555
556 #0 [postcondition]
557
558 #1 [postcondition]
559 559 #2 [postcondition]
560 560 #3 [postcondition]
561
562 562 #CVC5 1.3.1
563
564 #0 [postcondition]
565
566 #1 [postcondition]
567 567 #2 [postcondition]
568 568 #3 [postcondition]
569
570 570 #CVC5 1.3.1
571
572 #0 [postcondition]
573
574 #1 [postcondition]
575 575 #2 [postcondition]
576 576 #3 [postcondition]
577
578 578 #CVC5 1.3.1
579
580 #0 [postcondition]
581
582 #1 [postcondition]
583 583 #2 [postcondition]
584 584 #3 [postcondition]
585
586 586 #CVC5 1.3.1
587
588 #0 [postcondition]
589
590 #1 [postcondition]
591 591 #2 [postcondition]
592 592 #3 [postcondition]
593
594 594 #CVC5 1.3.1
595
596 #0 [postcondition]
597
598 #1 [postcondition]
599 599 #2 [postcondition]
600 600 #3 [postcondition]
601
602 602 #CVC5 1.3.1
603
604 #0 [postcondition]
605
606 #1 [postcondition]
607 607 #2 [postcondition]
608 608 #3 [postcondition]
609
610 610 #CVC5 1.3.1
611
612 #0 [postcondition]
613
614 #1 [postcondition]
615 615 #2 [postcondition]
616 616 #3 [postcondition]
617
618 618 #CVC5 1.3.1
619
620 #0 [postcondition]
621
622 #1 [postcondition]
623 623 #2 [postcondition]
624 624 #3 [postcondition]
625
626 626 #CVC5 1.3.1
627
628 #0 [postcondition]
629
630 #1 [postcondition]
631 631 #2 [postcondition]
632 632 #3 [postcondition]
633
634 634 #CVC5 1.3.1
635
636 #0 [postcondition]
637
638 #1 [postcondition]
639 639 #2 [postcondition]
640 640 #3 [postcondition]
641
642 642 #CVC5 1.3.1
643
644 #0 [postcondition]
645
646 #1 [postcondition]
647 647 #2 [postcondition]
648 648 #3 [postcondition]
649
650 650 #CVC5 1.3.1
651
652 #0 [postcondition]
653
654 #1 [postcondition]
655 655 #2 [postcondition]
656 656 #3 [postcondition]
657
658 658 #CVC5 1.3.1
659
660 #0 [postcondition]
661
662 #1 [postcondition]
663 663 #2 [postcondition]
664 664 #3 [postcondition]
665
666 666 #CVC5 1.3.1
667
668 #0 [postcondition]
669
670 #1 [postcondition]
671 671 #2 [postcondition]
672 672 #3 [postcondition]
673
674 674 #CVC5 1.3.1
675
676 #0 [postcondition]
677
678 #1 [postcondition]
679 679 #2 [postcondition]
680 680 #3 [postcondition]
681
682 682 #CVC5 1.3.1
683
684 #0 [postcondition]
685
686 #1 [postcondition]
687 687 #2 [postcondition]
688 688 #3 [postcondition]
689
690 690 #CVC5 1.3.1
691
692 #0 [postcondition]
693
694 #1 [postcondition]
695 695 #2 [postcondition]
696 696 #3 [postcondition]
697
698 698 #CVC5 1.3.1
699
700 #0 [postcondition]
701
702 #1 [postcondition]
703 703 #2 [postcondition]
704 704 #3 [postcondition]
705
706 706 #CVC5 1.3.1
707
708 #0 [postcondition]
709
710 #1 [postcondition]
711 711 #2 [postcondition]
712 712 #3 [postcondition]
713
714 714 #CVC5 1.3.1
715
716 #0 [postcondition]
717
718 #1 [postcondition]
719 719 #2 [postcondition]
720 720 #3 [postcondition]
721
722 722 #CVC5 1.3.1
723
724 #0 [postcondition]
725
726 #1 [postcondition]
727 727 #2 [postcondition]
728 728 #3 [postcondition]
729
730 730 #CVC5 1.3.1
731
732 #0 [postcondition]
733
734 #1 [postcondition]
735 735 #2 [postcondition]
736 736 #3 [postcondition]
737
738 738 #CVC5 1.3.1
739
740 #0 [postcondition]
741
742 #1 [postcondition]
743 743 #2 [postcondition]
744 744 #3 [postcondition]
745
746 746 #CVC5 1.3.1
747
748 #0 [postcondition]
749
750 #1 [postcondition]
751 751 #2 [postcondition]
752 752 #3 [postcondition]
753
754 754 #CVC5 1.3.1
755
756 #0 [postcondition]
757
758 #1 [postcondition]
759 759 #2 [postcondition]
760 760 #3 [postcondition]
761
762 762 #CVC5 1.3.1
763
764 #0 [postcondition]
765
766 #1 [postcondition]
767 767 #2 [postcondition]
768 768 #3 [postcondition]
769
770 770 #CVC5 1.3.1
771
772 #0 [postcondition]
773
774 #1 [postcondition]
775 775 #2 [postcondition]
776 776 #3 [postcondition]
777
778 778 #CVC5 1.3.1
779
780 #0 [postcondition]
781
782 #1 [postcondition]
783 783 #2 [postcondition]
784 784 #3 [postcondition]
785
786 786 #CVC5 1.3.1
787
788 #0 [postcondition]
789
790 #1 [postcondition]
791 791 #2 [postcondition]
792 792 #3 [postcondition]
793
794 794 #CVC5 1.3.1
795
796 #0 [postcondition]
797
798 #1 [postcondition]
799 799 #2 [postcondition]
800 800 #3 [postcondition]
801
802 802 #CVC5 1.3.1
803
804 #0 [postcondition]
805
806 #1 [postcondition]
807 807 #2 [postcondition]
808 808 #3 [postcondition]
809
810 810 #CVC5 1.3.1
811
812 #0 [postcondition]
813
814 #1 [postcondition]
815 815 #2 [postcondition]
816 816 #3 [postcondition]
817
818 818 #CVC5 1.3.1
819
820 #0 [postcondition]
821
822 #1 [postcondition]
823 823 #2 [postcondition]
824 824 #3 [postcondition]
825
826 826 #CVC5 1.3.1
827
828 #0 [postcondition]
829
830 #1 [postcondition]
831 831 #2 [postcondition]
832 832 #3 [postcondition]
833
834 834 #CVC5 1.3.1
835
836 #0 [postcondition]
837
838 #1 [postcondition]
839 839 #2 [postcondition]
840 840 #3 [postcondition]
841
842 842 #CVC5 1.3.1
843
844 #0 [postcondition]
845
846 #1 [postcondition]
847 847 #2 [postcondition]
848 848 #3 [postcondition]
849
850 850 #CVC5 1.3.1
851
852 #0 [postcondition]
853
854 #1 [postcondition]
855 855 #2 [postcondition]
856 856 #3 [postcondition]
857
858 858 #CVC5 1.3.1
859
860 #0 [postcondition]
861
862 #1 [postcondition]
863 863 #2 [postcondition]
864 864 #3 [postcondition]
865
866 866 #CVC5 1.3.1
867
868 #0 [postcondition]
869
870 #1 [postcondition]
871 871 #2 [postcondition]
872 872 #3 [postcondition]
873
874 874 #CVC5 1.3.1
875
876 #0 [postcondition]
877
878 #1 [postcondition]
879 879 #2 [postcondition]
880 880 #3 [postcondition]
881
882 882 #CVC5 1.3.1
883
884 #0 [postcondition]
885
886 #1 [postcondition]
887 887 #2 [postcondition]
888 888 #3 [postcondition]
889
890 890 #CVC5 1.3.1
891
892 #0 [postcondition]
893
894 #1 [postcondition]
895 895 #2 [postcondition]
896 896 #3 [postcondition]
897
898 898 #CVC5 1.3.1
899
900 #0 [postcondition]
901
902 #1 [postcondition]
903 903 #2 [postcondition]
904 904 #3 [postcondition]
905
906 906 #CVC5 1.3.1
907
908 #0 [postcondition]
909
910 #1 [postcondition]
911 911 #2 [postcondition]
912 912 #3 [postcondition]
913
914 914 #CVC5 1.3.1
915
916 #0 [postcondition]
917
918 #1 [postcondition]
919 919 #2 [postcondition]
920 920 #3 [postcondition]
921
922 922 #CVC5 1.3.1
923
924 #0 [postcondition]
925
926 #1 [postcondition]
927 927 #2 [postcondition]
928 928 #3 [postcondition]
929
930 930 #CVC5 1.3.1
931
932 #0 [postcondition]
933
934 #1 [postcondition]
935 935 #2 [postcondition]
936 936 #3 [postcondition]
937
938 938 #CVC5 1.3.1
939
940 #0 [postcondition]
941
942 #1 [postcondition]
943 943 #2 [postcondition]
944 944 #3 [postcondition]
945
946 946 #CVC5 1.3.1
947
948 #0 [postcondition]
949
950 #1 [postcondition]
951 951 #2 [postcondition]
952 952 #3 [postcondition]
953
954 954 #CVC5 1.3.1
955
956 #0 [postcondition]
957
958 #1 [postcondition]
959 959 #2 [postcondition]
960 960 #3 [postcondition]
961
962 962 #CVC5 1.3.1
963
964 #0 [postcondition]
965
966 #1 [postcondition]
967 967 #2 [postcondition]
968 968 #3 [postcondition]
969
970 970 #CVC5 1.3.1
971
972 #0 [postcondition]
973
974 #1 [postcondition]
975 975 #2 [postcondition]
976 976 #3 [postcondition]
977
978 978 #CVC5 1.3.1
979
980 #0 [postcondition]
981
982 #1 [postcondition]
983 983 #2 [postcondition]
984 984 #3 [postcondition]
985
986 986 #CVC5 1.3.1
987
988 #0 [postcondition]
989
990 #1 [postcondition]
991 991 #2 [postcondition]
992 992 #3 [postcondition]
993
994 994 #CVC5 1.3.1
995
996 #0 [postcondition]
997
998 #1 [postcondition]
999 999 #2 [postcondition]
1000 1000 #3 [postcondition]
1001
1002 1002 #CVC5 1.3.1
1003
1004 #0 [postcondition]
1005
1006 #1 [postcondition]
1007 1007 #2 [postcondition]
1008 1008 #3 [postcondition]
1009
1010 1010 #CVC5 1.3.1
1011
1012 #0 [postcondition]
1013
1014 #1 [postcondition]
1015 1015 #2 [postcondition]
1016 1016 #3 [postcondition]
1017
1018 1018 #CVC5 1.3.1
1019
1020 #0 [postcondition]
1021
1022 #1 [postcondition]
1023 1023 #2 [postcondition]
1024 1024 #3 [postcondition]
1025
1026 1026 #CVC5 1.3.1
1027
1028 #0 [postcondition]
1029
1030 #1 [postcondition]
1031 1031 #2 [postcondition]
1032 1032 #3 [postcondition]
1033
1034 1034 #CVC5 1.3.1
1035
1036 #0 [postcondition]
1037
1038 #1 [postcondition]
1039 1039 #2 [postcondition]
1040 1040 #3 [postcondition]
1041
1042 1042 #CVC5 1.3.1
1043
1044 #0 [postcondition]
1045
1046 #1 [postcondition]
1047 1047 #2 [postcondition]
1048 1048 #3 [postcondition]
1049
1050 1050 #CVC5 1.3.1
1051
1052 #0 [postcondition]
1053
1054 #1 [postcondition]
1055 1055 #2 [postcondition]
1056 1056 #3 [postcondition]
1057
1058 1058 #CVC5 1.3.1
1059
1060 #0 [postcondition]
1061
1062 #1 [postcondition]
1063 1063 #2 [postcondition]
1064 1064 #3 [postcondition]
1065
1066 1066 #CVC5 1.3.1
1067
1068 #0 [postcondition]
1069
1070 #1 [postcondition]
1071 1071 #2 [postcondition]
1072 1072 #3 [postcondition]
1073
1074 1074 #CVC5 1.3.1
1075
1076 #0 [postcondition]
1077
1078 #1 [postcondition]
1079 1079 #2 [postcondition]
1080 1080 #3 [postcondition]
1081
1082 1082 #CVC5 1.3.1
1083
1084 #0 [postcondition]
1085
1086 #1 [postcondition]
1087 1087 #2 [postcondition]
1088 1088 #3 [postcondition]
1089
1090 1090 #CVC5 1.3.1
1091
1092 #0 [postcondition]
1093
1094 #1 [postcondition]
1095 1095 #2 [postcondition]
1096 1096 #3 [postcondition]
1097
1098 1098 #CVC5 1.3.1
1099
1100 #0 [postcondition]
1101
1102 #1 [postcondition]
1103 1103 #2 [postcondition]
1104 1104 #3 [postcondition]
1105
1106 1106 #CVC5 1.3.1
1107
1108 #0 [postcondition]
1109
1110 #1 [postcondition]
1111 1111 #2 [postcondition]
111
```

## 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  $(\text{termA} \neq \text{termB}) \wedge (\text{termA} \neq \text{termC}) \wedge (\text{termB} \neq \text{termC})$ .

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  $h$ .<sup>12</sup>

To verify the code above, execute the following command:

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

```

Status Theories/Goals
File Edit Tools View Help
Status Task ex_oneway_ANOVA.mlw atomsForums.mlw
Time 12
13 let ex_oneway_ANOVA (d1 d2 d3 : dataset real)
14 (* Executes oneway ANOVA for 3 population means *)
15 requires { for_all (fun d -> d.scale = Interval) (Cons d1
16 independent_list (Cons d1 (Cons d2 (Cons d3 Nil)))
17 is_empty lst /\ for_all2
18 (fun p y -> sampled y p)
19 (Cons p1 (Cons p2 (Cons p3 Nil))), /\
20 (Cons d1 (Cons d2 (Cons d3 Nil))), /\
21 ((World !st interp) |= Possible (t_m1 $< t_m2))
22 ((World !st interp) |= Possible (t_m1 $> t_m2))
23 ((World !st interp) |= Possible (t_m1 $< t_m3))
24 ((World !st interp) |= Possible (t_m1 $> t_m3))
25 ((World !st interp) |= Possible (t_m2 $< t_m3))
26 ((World !st interp) |= Possible (t_m2 $> t_m3))
27 ((World !st interp) |= Possible (t_m2 $> t_m3))
28 }
29 ensures {
30   let p = result in
31   let h = (t_m1 $!= t_m2) $|| (t_m1 $!= t_m3) $|| (t_m2 $|| (t_m2 $&& Eq p = compose_pvs h !st &&
32   Eq p = StatB (Eq p) h
33   (World !st interp) |= StatB (Eq p) h
34 )
35
0/0/0 type commands here

```

Messages Log Interpreter Power output Counterexample

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

```

<sup>12</sup>  $\$&&$  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
```

```

File Edit Tools View Help
Status Theories/Goals Time Task ex_alexandergovern.mlw atomsFormulas.mlw
14 let ex_alexandergovern (d1 d2 d3 d4 : dataset real)
15 (* Executes Alexander-Govern test for 3 population means *)
16 requires { independent_list (Cons d1 (Cons d2 (Cons d3 (
17 for_all (fun d -> d.scale = Interval) (Cons d
18 is_empty !st /\
19 for_all2
20
21 (fun p y -> sampled y p)
22 (Cons p1 (Cons p2 (Cons p3 (Cons p4 Nil))))
23 (Cons d1 (Cons d2 (Cons d3 (Cons d4 Nil))))))
24 ((World !st interp) |= Possible (t_m1 $< t_m2)
25 ((World !st interp) |= Possible (t_m1 $> t_m2)
26 ((World !st interp) |= Possible (t_m1 $< t_m3)
27 ((World !st interp) |= Possible (t_m1 $> t_m3)
28 ((World !st interp) |= Possible (t_m1 $< t_m4)
29 ((World !st interp) |= Possible (t_m1 $> t_m4)
30 ((World !st interp) |= Possible (t_m2 $< t_m3)
31 ((World !st interp) |= Possible (t_m2 $> t_m3)
32 ((World !st interp) |= Possible (t_m2 $< t_m4)
33 ((World !st interp) |= Possible (t_m2 $> t_m4)
34 ((World !st interp) |= Possible (t_m3 $< t_m4)
35 ((World !st interp) |= Possible (t_m3 $> t_m4)
36
0/0/0 type commands here

```

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)) /\

```

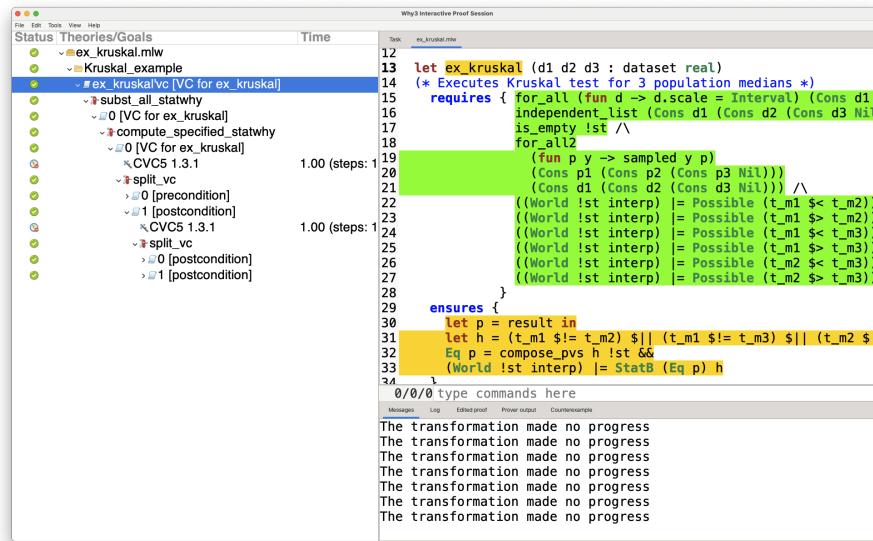
```

        ((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
```



## 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
```

The screenshot shows the Why3 Interactive Proof Session window. On the left, there is a tree view of theories and goals under 'Status Theories/Goals'. One goal is expanded, showing its proof steps and associated SMT solvers (CVC5 1.3.1). On the right, the main area displays OCaml code for a Tukey HSD test. The code includes imports for `CameleerBHL` and `Tukey\_HSD`, defines a module `Example6\_Tukey\_HSD` with a struct type, and contains several let-bindings for normal distributions and array operations. The code editor has syntax highlighting for OCaml. Below the code editor, the status bar shows 'The transformation made no progress' and '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 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  $\chi^2$  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](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](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](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](https://doi.org/10.1007/978-3-030-81688-9_31)

Names of testing methods	Goals of the hypothesis testing	File names	Names in <code>scipy.stats</code>
<i>t</i> -test	mean of a population two population means (independent samples) two population means (paired comparisons)	ttest.mlw ttest.mlw ttest.mlw	ttest_1samp ttest_ind ttest_rel
$\chi^2$ -test	goodness of fit independence of two categorical data	chisquare.mlw chi2_contingency.mlw	chisquare chi2_contingency
<i>F</i> -test		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	
Williams' test	comparisons of means against a control group	williams.mlw	
Steel test	comparisons of means against a control group	steel.mlw	
Kruskal-Wallis <i>H</i> -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	friedmanchi-square.mlw	friedmanchi-square
<i>k</i> -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	$\chi^2$ 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 <i>t</i> -test	0.46
	Student's <i>t</i> -test	0.48
	Welch's <i>t</i> -test	0.43
	Paired <i>t</i> -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.

Test methods	Metric	#groups					
		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