

User Documentation for StatWhy v.1.3.0

Yusuke Kawamoto¹, Kentaro Kobayashi^{1,2}, and Kohei Suenaga³ *

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 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: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_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 true⁸. 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 → 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_1 ∨ 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_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 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 κ_\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 φ with a p -value $\alpha \leq 0.05$. Since the result of the hypothesis test may be wrong, we use the belief modality $\mathbf{K}_{y,A}^{\leq 0.05}$ 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 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.

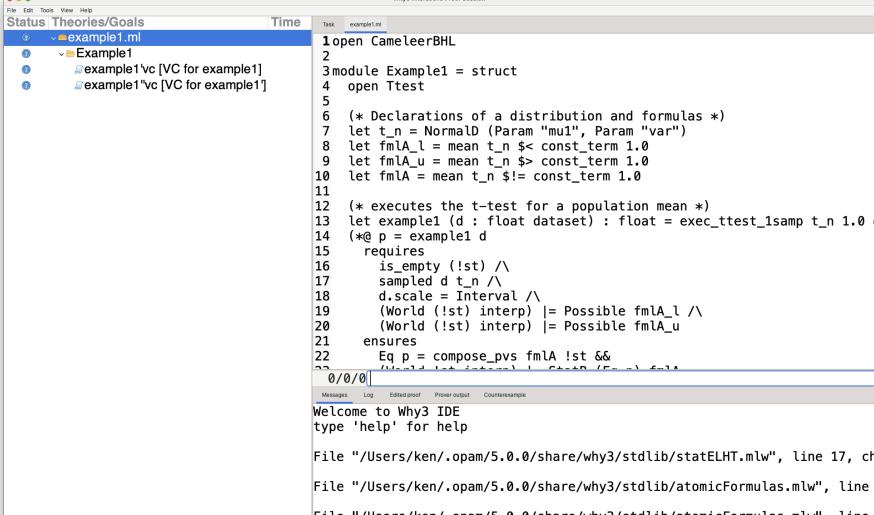
6.1 Simple *t*-test (One-Sample *t*-test)

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.



The screenshot shows the Why3 IDE interface. The left pane displays the 'Theories/Goals' tree, which includes a project named 'example1.ml' containing an 'Example1' module with two verification conditions ('example1vc [VC for example1]' and 'example1"vc [VC for example1]'). The right pane shows the 'Task' editor with the following OCaml code:

```

open CameleerBHL
2
3module 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_1samp t_n 1.0
14 (*@ p = example1 d
15  requires
16  is_empty (!st) /\ sampled d t_n /\ d.scale = Interval /\ 
17  (World (!st) interp) |= Possible fmlA_l /\ 
18  (World (!st) interp) |= Possible fmlA_u
19  ensures
20  Eq p = compose_pvs fmlA !st &&
21  (World (!st) interp) |= SAMP (Eq p) fmlA
22 0/0/0

```

The status bar at the bottom indicates: 'Welcome to Why3 IDE type 'help' for help'. Below that, it lists the loaded files: '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', and '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: `sampler 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.

```

Status Theories/Goals
Task example1.ml
1 ✓ example1.ml
2 ✓ Example1
3 ✓ example1'vc [VC for example1]
4 ✓ example1''vc [VC for example1]

Time
20
21 ensures

```

The screenshot shows the Why3 IDE interface. The left panel displays the theories and goals, with several items expanded. The right panel shows the task details for the selected goal, which is `example1'vc [VC for example1]`. The task panel includes a dropdown menu for selecting a tactic, a list of available tactics, and a command-line interface for entering commands. The number 20 is at the bottom left, and 21 ensures is at the bottom right.

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:

```

Status Theories/Goals
Task example1.ml
1 ✓ example1.ml
2 ✓ Example1
3 ✓ example1'vc [VC for example1]
4 ✓ example1''vc [VC for example1]
5 ✓ #0 [VC for example1]
6 ✓ #1 [VC for example1]
7 ✓ #2 [VC for example1]

Time
20
21 ensures

```

This screenshot shows the Why3 IDE after the goal `#0 [VC for example1]` has been discharged successfully. The checkmark (✓) is now present next to the goal in the left panel. The right panel shows the task details for the goal, with the command `exec t-test` highlighted. The number 20 is at the bottom left, and 21 ensures is at the bottom right.

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 interface. On the left, there's a tree view of the theories and goals. The main area displays the proof state with various tactics and their sub-tactics, including `d` as a dataset real, `H2` as an interval, and `H1` and `H2` as world terms. The proof state includes sections for `CVC5 1.3.1`, `Goal`, and `goal example1` with its own sub-tactics. At the bottom, there are tabs for `Messages`, `Log`, `Edited proof`, `Printer output`, and `Counterexample`. A message at the bottom states: "The transformation made no progress 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 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)

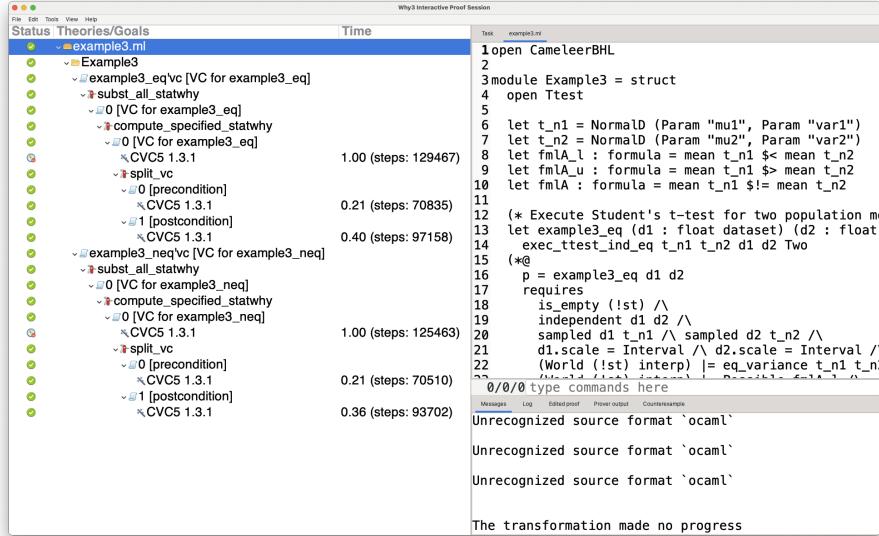
```

```

| 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_{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 \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 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`.

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:

The screenshot shows the Why3 Interactive Proof Session interface. The left pane displays the theory and goals, including the `example5` module and its sub-components like `Example5`, `example5vc`, and various proof obligations. The right pane shows the OCaml code for the `example5` module, with annotations indicating verification status (e.g., 1.00, 5.00, 60.00) and specific error messages. A prominent error message at the bottom right states: "The transformation made no progress Unrecognized source format 'ocaml'".

```

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_ttest_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  0/0/0 type commands here
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
678
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
827
828
829
829
830
831
832
833
834
835
836
837
837
838
839
839
840
841
842
843
844
845
846
846
847
848
848
849
849
850
851
852
853
854
855
856
856
857
858
858
859
859
860
861
862
863
864
865
866
866
867
868
868
869
869
870
871
872
873
874
875
876
876
877
878
878
879
879
880
881
882
883
884
885
886
886
887
888
888
889
889
890
891
892
893
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
904
905
905
906
907
907
908
908
909
909
910
911
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1
```

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 .¹²

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

```

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
```

```

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
```

```

File Edit Tools View Help
Status Theories/Goals Time Task ex_kruskal.mlw
12
13 let ex_kruskal (d1 d2 d3 : dataset real)
14 (* Executes Kruskal test for 3 population medians *)
15 requires { for_all (fun d -> d.scale = Interval) (Cons d1 (
16 independent_list (Cons d1 (Cons d2 (Cons d3 Nil))
17 is_empty !st) /\
18 for_all2
19     (fun p y -> sampled y p)
20     (Cons p1 (Cons p2 (Cons p3 Nil)))
21     (Cons d1 (Cons d2 (Cons d3 Nil))) /\
22     ((World !st interp) |= Possible (t_m1 $< t_m2))
23     ((World !st interp) |= Possible (t_m1 $> t_m2))
24     ((World !st interp) |= Possible (t_m1 $< t_m3))
25     ((World !st interp) |= Possible (t_m1 $> 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 $!=
32   Eq p = compose_pvs h !st &&
33   (World !st interp) |= StatB (Eq p) h
34 }
0/0/0 type commands here
Messages Log Editor/proof Prove output Counterexample
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
```

The screenshot shows the Why3 Interactive Proof Session window. On the left, a tree view displays the project structure with files like `example6.ml`, `atomsFormulas.ml`, and `tukey_HSD.ml`. A specific goal, `#example6_tukey_hsd`vc [VC for example6_tukey_hsd]`, is selected. On the right, the code editor contains OCaml code for a Tukey HSD test. The code includes imports for `CamelotBHL` and `Tukey_HSD`, defines a module `Example6_Tukey_HSD`, and implements a function `exec_tukey_hsd` that takes three arrays of floats (`t_n1`, `t_n2`, `t_n3`) and a float array (`d1`, `d2`, `d3`). The code also includes annotations for precondition and postcondition formulas. The status bar at the bottom indicates that the transformation made no progress and shows unrecognized source format errors.

```

File Edit Tools View Help
Status Theories/Goals
example6.ml atomsFormulas.ml tukey_HSD.ml
Time
Task example6.ml atomsFormulas.ml tukey_HSD.ml
1 open CamelotBHL
2
3 module Example6_Tukey_HSD = struct
4   open Tukey_HSD
5   open Array
6
7   let t_n1 = NormalD (Param "mu1", Param "var")
8   let t_n2 = NormalD (Param "mu2", Param "var")
9   let t_n3 = NormalD (Param "mu3", Param "var")
10
11  let terms3 = [ mean t_n1; mean t_n2; mean t_n3 ]
12
13 (* Execute Tukey's HSD test for multiple comparison of 3 groups
14 let example6_tukey_hsd d1 d2 d3 : float array =
15   exec_tukey_hsd [ t_n1; t_n2; t_n3 ] [ d1; d2; d3 ]
16 (*@
17   ps = example6_tukey_hsd d1 d2 d3
18   requires
19     is_empty (!st) /\ 
20     independent_list (Cons d1 (Cons d2 (Cons d3 Nil))) /\ 
21     for_all2
22     sampled
23   *)
24
25 /0/0 type commands here
Messages Log Edited proof Power output Counterexamples
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 <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
χ^2 -test	goodness of fit independence of two categorical data	chisquare.mlw chi2_contingency.mlw	chisquare chi2_contingency
<i>F</i> -test		fstat.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	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.

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