

# SpecForge User Guide

*SpecForge* is an AI-powered formal specification authoring tool based on *Lilo*, a domain specific language designed for specifying temporal systems.

This guide covers installation, the *Lilo* specification language, the Python SDK, and using Lilo with VSCode.

To get started, see [setting up](#), releases are available from the [releases page](#).

Other versions of this guide:

- In [PDF format](#).
- In [Japanese](#). (このガイドには日本語版もあります.)

# Setting up SpecForge

The SpecForge suite consists of a few components.

- The **SpecForge Server** which is the backend server which the other components connects to. It is usually distributed as a Docker image.
- The **SpecForge VSCode Extension** which provides Lilo Language support in VSCode for editing and managing specifications, as well as rendering interactive visualizations.
- The **SpecForge Python SDK** which provides an API for interacting with the SpecForge server from Python code. This can be used to communicate and exchange specifications or data with the SpecForge server from Python scripts or Jupyter notebooks.

All necessary files can be obtained from the [SpecForge releases](#) page. Usually you should download and extract the complete release ( `vx.y.z-complete.zip` ) and perform commands in the resulting directory.

# Setting up the SpecForge Server

## Obtaining the Docker Image and Running the Server

The SpecForge Server is distributed as a Docker Image via GHCR (GitHub Container Registry). The recommended way to obtain and run the Docker Image is through Docker Compose. You can find the latest `docker-compose-x.y.z.yml` file on the official [Release Page](#).

With `docker-compose-x.y.z.yml` in current directory run:

```
docker compose -f /path/to/docker-compose-x.y.z.yml up --abort-on-container-exit
```

---

The flag `--abort-on-container-exit` is recommended so that the container fails fast on startup errors.

---

Be sure to update the path above to point to your actual `docker-compose-x.y.z.yml` file (where the `x.y.z` corresponds to the version number).

You can verify that the server is up by navigating to <http://localhost:8080/health>, which should show the appropriate version information.

---

You need to obtain and configure a valid license for the server to start successfully. Please refer to the [Licensing and Configuration](#) guide for more details.

---

## Updating the Image (upon new release)

From time to time, we release new versions of the SpecForge Server. To use the latest version, you could use the updated `docker-compose-x.y.z.yml` file from the [Release Page](#).

Another option would be to set the `image` field in your Docker Compose file to point to the latest version, e.g.,

```
image: ghcr.io/imiron-io/specforge/specforge-backend:latest
```

If you have done so, you can simply run `pull` to pull the latest image:

```
docker compose -f /path/to/docker-compose.yml pull
```

# Licensing and Configuration

The provided `docker-compose.yml` file is configured to run the SpecForge server.

By default, the SpecForge server runs on port `8080`.

## Licensing

The server requires a valid license. At startup, the license is checked and the server will exit early if the license is missing or invalid.

If you do not have a license, please contact the SpecForge team to request one through [form](#).

When running the SpecForge via docker containers, the license file must be made available inside the container. To do so,

1. Place your `license.json` file in a known location on your host machine (for example `~/.config/specforge/license.json`).
2. Modify the following lines of your `docker-compose.yml`:

```
- type: bind
  source: ~/.config/specforge/license.json # make sure this path points to the saved
  license file on your host machine
  target: /app/license.json # do not change this path
  read_only: true
```

## LLM Provider Configuration

For using LLM-based features such as natural-language based spec generation and error explanation, you need to configure the LLM provider. Currently, we support OpenAI, Ollama and Gemini. API keys for OpenAI and Gemini can be obtained from [the OpenAI website](#) and [the Gemini API website](#) respectively. [Ollama](#) is a framework for running and serving an LLM locally.

To configure the LLM provider, replace the following environment variables in the `docker-compose.yml` file with appropriate values. For Gemini and OpenAI, the model is set to `gemini-2.5-flash` and `gpt-5-nano-2025-08-07` respectively by default, if the `LLM_MODEL` variable is not set. The user must set the `LLM_MODEL` variable when using Ollama. The `OLLAMA_API_BASE` variable should be changed if Ollama is running on a remote machine. The `OPENAI_API_KEY` or `GEMINI_API_KEY` variable is needed for OpenAI or Gemini.

```
- LLM_PROVIDER=openai # other options: ollama, gemini
- LLM_MODEL=gpt-5-nano-2025-08-07 # choose the appropriate model for your provider, to
  change from default
- OPENAI_API_KEY=${OPENAI_API_KEY}
- GEMINI_API_KEY=${GEMINI_API_KEY}
- OLLAMA_API_BASE=http://127.0.0.1:11434 # change if your ollama server is running
  remotely
```

Without an appropriate LLM provider configuration, LLM-based SpecForge features would be unavailable.

It is possible to insert API keys directly, e.g.:

```
- LLM_PROVIDER=gemini
- GEMINI_API_KEY=abc123XYZ # no string quotes
```

but it is better to use environment variables.

# VSCoDe Extension

The Lilo Language Extension for VSCoDe provides

- Syntax Highlighting, Typechecking and Autocompletion for `.lilo` files
- Satisfiability and Redundancy checking for specifications
- Support for visualizing monitoring results in Python notebooks

## Installation

To install the SpecForge VSCoDe extension, locate the `lilo-language-x.x.x.vsix` file (included in [releases](#)), and install it in one of the following ways.

- Open VSCoDe's extensions tab (Ctrl+Shift+X or Cmd+Shift+X), click on the three dots at the top right, and select `Install from VSIX...`
- Open VSCoDe's command palette (Ctrl+Shift+P or Cmd+Shift+P), type `Extensions: Install from VSIX...`, and select the `.vsix` file

## Usage and Configuration

- For the extension to work, the SpecForge server must be running (see above).
- The URI corresponding to the server can be configured if necessary using the [extension settings](#). Do not add a trailing slash at the end of this URL.

Once the extension is installed and the server is running, it should automatically be working on `.lilo` files, and in relevant Python notebooks.

# Setting up the Python SDK

The Python SDK is a python library which can be used to interact with SpecForge tools programmatically from within Python programs, including notebooks.

The Python SDK is packaged as a wheel file with the name `specforge_sdk-x.x.x-py3-none-any.whl`.

Refer to the [SpecForge Python SDK](#) guide for an overview of the SDK features and capabilities.

## A Sample Walkthrough

The Python SDK can be installed directly using `pip`, or defined as a dependency via a build environment such as `poetry` or `uv`.

We discuss below how such an environment can be setup using `uv`. If you prefer to use a different build system, the workflow should be similar.

1. Install `uv` on your operating system. See the [uv installation guide](#) for more details.
2. Create a new project directory and navigate into it. Populate it with a `pyproject.toml` file.
3. Declare the dependencies in the `pyproject.toml` file.
  - The wheel file for the Python SDK can be declared as a local dependency. Ensure that a correct path to the wheel file is provided.
  - Features of SpecForge, such as the interactive monitor, can be used as a part of Python Notebooks. To do so, you may want to include `jupyterlab` as a dependency as well.
  - Libraries such as `numpy`, `pandas` and `matplotlib` are frequently included for data processing and visualization.
  - Here is an example `pyproject.toml` file:

```
[project]
name = "sample-project"
version = "0.1.0"
description = "Sample Project for Testing SpecForge SDK"
authors = [{ name = "Imiron Developers", email = "info@imiron.io" }]
readme = "README.md"
requires-python = ">=3.12"
dependencies = [
    "jupyterlab>=4.4.5",
    "pandas>=2.3.1",
    "matplotlib>=3.10.3",
    "numpy>=2.3.2",
    "specforge-sdk",
]

[tool.uv.sources]
specforge_sdk = { path = "lib/specforge_sdk-0.5.4.dev5-py3-none-any.whl" }
```

4. Run `uv sync`. This should create a `.venv` directory which would have the appropriate dependencies (including the correct version of python) installed.
5. Run `source .venv/bin/activate` to use the Shell Hook with access to `python`. You can confirm that this has been configured correctly as follows.

```
```bash
$ source .venv/bin/activate
(falsification-examples) $ which python
/Users/agnishom/code/reqeng/api/examples/projects/falsification/.venv/bin/python
```
```

6. Now, you can browse the example notebooks. Make sure that your notebook is connected to the kernel in the `.venv`. This is usually configured automatically, but can also be done manually. To do so, run `jupyter server` and copy and paste the server URL in the kernel settings in the VSCode notebook viewer.

# Project Configuration

Lilo projects can use an optional `lilo.toml` at the project root. If the file or any of its fields are missing, sensible defaults apply. The Python SDK and the VS Code extension read this file and apply the semantics accordingly.

Config controls:

- Project name and source path
- Language behavior (interval mode, freeze)
- Diagnostics (consistency, redundancy, optimize, unused defs) and their timeouts
- Optional registry of `system_falsifier` entries.

Below are the schema and defaults, followed by a complete example.

## Schema and defaults

Top-level keys and their defaults when omitted:

- `project`
  - `name` (string).
    - Default: `""`.
    - On init: set to the provided name; otherwise to the name of the project root directory.
  - `source` (path string). Default: `"src/"`
- `language`
  - `interval.mode` (string). Supported: `"static"`. Default: `"static"`
  - `freeze.enabled` (bool). Default: `true`
- `diagnostics`
  - `consistency.enabled` (bool). Default: `true`
  - `consistency.timeouts.named` (seconds, float). Default: `0.5`
  - `consistency.timeouts.system` (seconds, float). Default: `1.0`
  - `redundancy.enabled` (bool). Default: `true`
  - `redundancy.timeouts.named` (seconds, float). Default: `0.5`
  - `redundancy.timeouts.system` (seconds, float). Default: `1.0`
  - `optimize.enabled` (bool). Default: `true`
  - `unused_defs.enabled` (bool). Default: `true`
- `[[system_falsifier]]` (array of tables, optional)
  - Each entry: `name` (string), `system` (string), `script` (string)
  - If absent or empty, the key is omitted from the file and treated as an empty list

Default file:

```
[project]
name = ""
source = "src/"
```

## Example `lilo.toml`

An example project with overrides.

```
[project]
name = "my-specs"
source = "src/"

[language]
freeze.enabled = true
interval.mode = "static"

[diagnostics.consistency]
enabled = true

[diagnostics.consistency.timeouts]
named = 5.0
system = 10.0

[diagnostics.optimize]
enabled = true

[diagnostics.redundancy]
enabled = false

[diagnostics.unused_defs]
enabled = false

[[system_falsifier]]
name = "Psitaliro ClimateControl Falsifier"
system = "climate_control"
script = "falsifiers/falsify_climate_control.py"

[[system_falsifier]]
name = "Psitaliro ALKS falsifier"
system = "lane_keeping"
script = "falsifiers/alks.py"
```



# A Whirlwind Tour

This section is a quick introduction to SpecForge's main capabilities through a hands-on example. We'll explore how to write specifications in the Lilo language and analyze them using SpecForge's VSCode extension.

## The Lilo Language: A Brief Introduction

Lilo is an expression-based temporal specification language designed for hybrid systems. Here are the key concepts:

**Primitive Types:** `Bool`, `Int`, `Float`, and `String`

**Operators:** Standard arithmetic (`+`, `-`, `*`, `/`), comparisons (`==`, `<`, `>`, etc.), and logical operators (`&&`, `||`, `=>`)

**Temporal Operators:** Lilo's distinguishing feature is its rich set of temporal logic operators:

- `always  $\phi$` :  $\phi$  is true at all future times
- `eventually  $\phi$` :  $\phi$  is true at some future time
- `past  $\phi$` :  $\phi$  was true at some past time
- `historically  $\phi$` :  $\phi$  was true at all past times

These operators can be qualified with time intervals, e.g., `eventually[0, 10]  $\phi$`  means  $\phi$  becomes true within 10 time units. More operators [are available](#).

**Systems:** Lilo specifications are organized into systems that group together:

- `signal s`: Time-varying input values (e.g., `signal temperature: Float`)
- `param s`: Non-temporal parameters that are not time-varying (e.g., `param max_temp: Float`)
- `type s`: Custom types for structured data
- `def initions`: Reusable definitions and helper functions
- `spec ifications`: Requirements that should hold for the system

A system file begins with a system declaration like `system temperature_control` and contains all the declarations for that system.

For a comprehensive guide to the language, see the [Lilo Language](#) chapter.

## Running Example

We'll use a temperature control system as our running example. This example project is available in the [releases](#). The system monitors temperature and humidity sensors, with specifications ensuring values remain within safe ranges:

```

system temperature_sensor

// Temperature Monitoring specifications
// This spec defines safety requirements for a temperature sensor system

import util use { in_bounds }

signal temperature: Float
signal humidity: Float

param min_temperature: Float
param max_temperature: Float

#[disable(redundancy)]
spec temperature_in_bounds = in_bounds(temperature, min_temperature, max_temperature)

spec always_in_bounds = always temperature_in_bounds

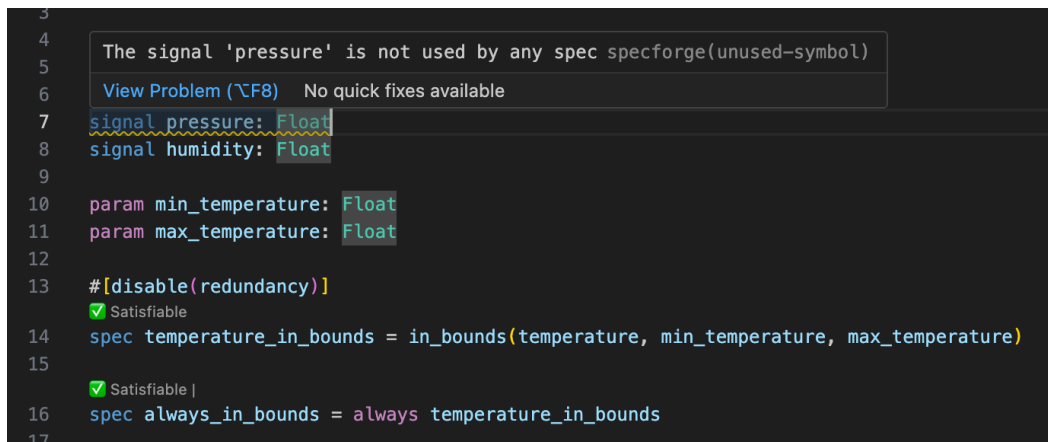
// Humidity should be reasonable when temperature is in normal range
spec humidity_correlation = always (
  (temperature >= 15.0 && temperature <= 35.0) =>
  (humidity >= 20.0 && humidity <= 80.0)
)

// Emergency condition - temperature exceeds critical thresholds
spec emergency_condition = temperature < 5.0 || temperature > 45.0

// Recovery specification - after emergency, system should stabilize
spec recovery_spec = always (
  emergency_condition =>
  eventually[0, 10] (temperature >= 15.0 && temperature <= 35.0)
)

```

The [VSCode extension](#) provides support for writing Lilo code, syntax highlighting, type-checking, warnings, spec satisfiability, etc.:



## Spec Analysis

Once you've written specifications for your system, the SpecForge VSCode extension provides various analysis capabilities:

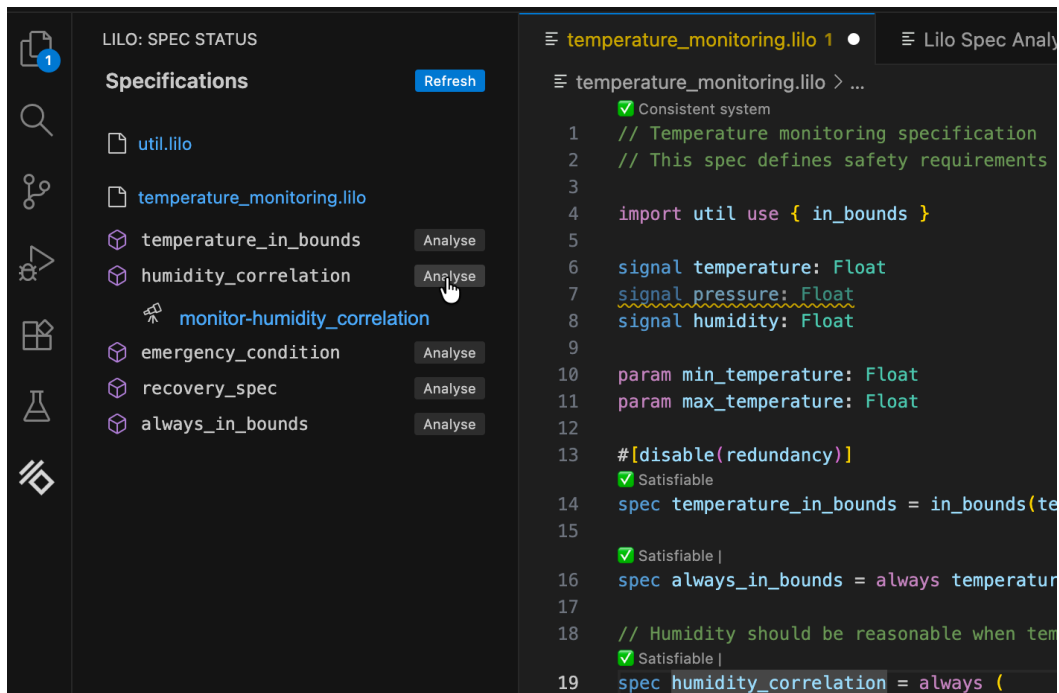
- **Monitor:** Check whether recorded system behavior satisfies specifications
- **Exemplify:** Generate example traces that satisfy specifications
- **Falsify:** Search for counterexamples that violate specifications, relative to some model
- **Export:** Convert specifications to other formats ( .json , .lilo , etc.)
- **Animate:** Visualize specification behavior over time

This can be done directly from within VSCode, or from within in a Jupyter notebook using the [Python SDK](#). We will perform analyses directly in VSCode here. The [VSCode guide](#) details all features in greater depth.

## Monitoring

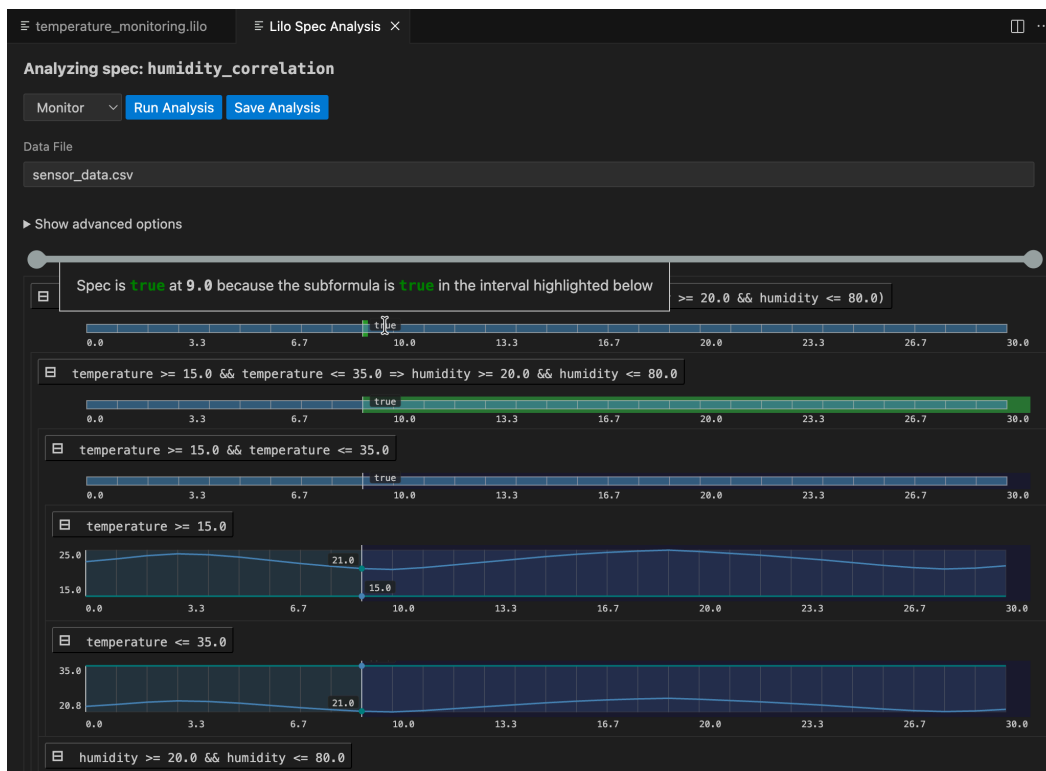
Monitoring checks whether actual system behavior, recorded in a data file, satisfies your specifications. You provide recorded trace data, and SpecForge evaluates a specification against it.

Navigate to the spec selection screen, and click the `Analyse` button for the spec you want to monitor.



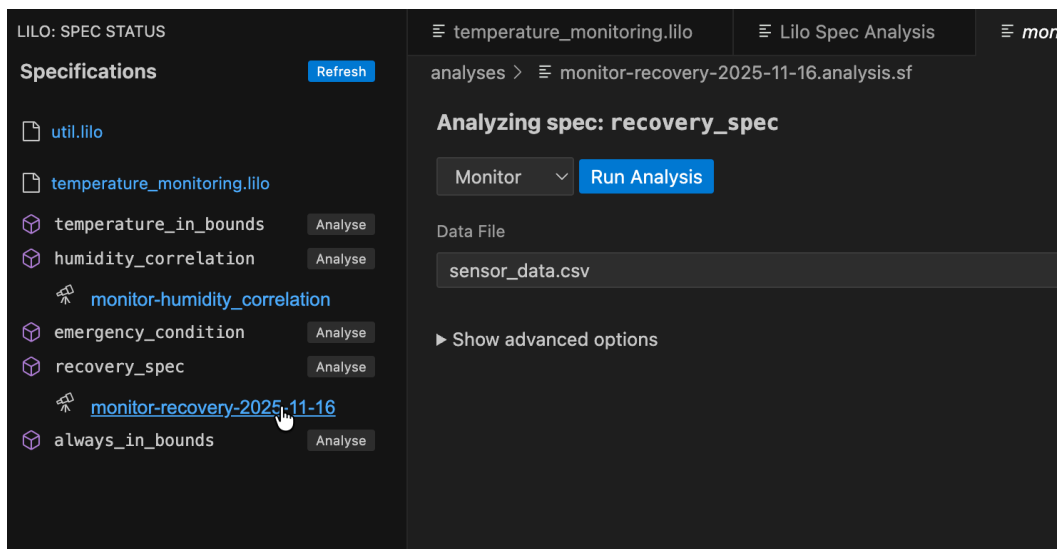
The screenshot shows the 'LILO: SPEC STATUS' window. On the left, a sidebar contains icons for file management, search, and analysis. The main area is titled 'Specifications' and lists several specifications: `util.lilo`, `temperature_monitoring.lilo`, `temperature_in_bounds`, `humidity_correlation`, `monitor-humidity_correlation`, `emergency_condition`, `recovery_spec`, and `always_in_bounds`. Each specification has an 'Analyse' button next to it. The `humidity_correlation` button is highlighted with a mouse cursor. On the right, a panel titled 'temperature\_monitoring.lilo 1' shows the analysis results for the selected specification. The results are displayed as a list of checks, each with a green checkmark and the word 'Satisfiable'. The checks include: 'Consistent system', 'Satisfiable', 'Satisfiable |', and 'Satisfiable |'. The specification code is also visible in the background.

After selecting a data file from the dropdown menu, click `Run Analysis`. The result is an analysis monitoring tree for the specification:



The result for the whole specification is shown at the top. Below this, you can drill down into sub-expressions of the specification, to understand what makes the spec true or false at any given time. Hovering over any of the signals will show a popup with an explanation of the result at that point in time, and will highlight relevant segments of sub-expression result signals.

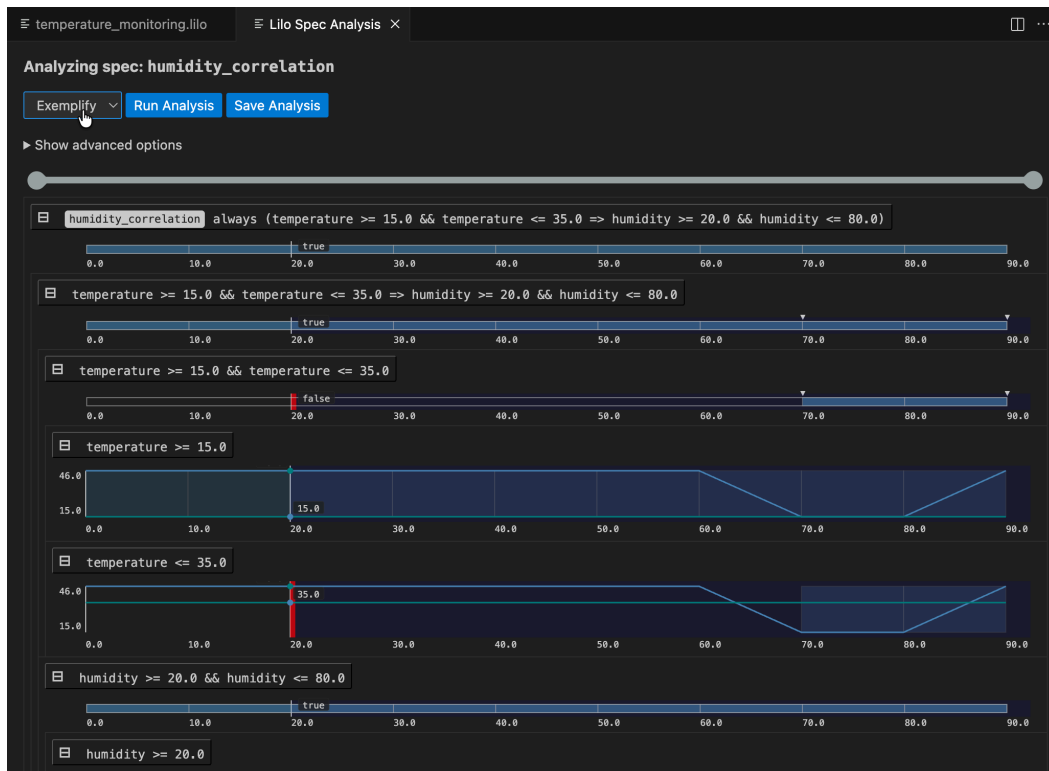
An analysis can be saved. To do so, click the **Save Analysis** button, and choose a location to save the analysis. You can then navigate to this analysis file and open it again in VSCode. The analysis will also show up in the specification status menu, under the relevant spec.



## Exemplification

The **Exemplify** analysis generates example traces that demonstrate satisfying behavior. This is useful for:

- Understanding what valid system behavior looks like
- Testing other components with realistic data
- Creating animations



If the exemplified data does not behave as expected, the specification might be wrong and need to be corrected. Exemplification can thus be used as an aid when authoring specifications.

## Falsification

If a model for the system is available, falsification can be used to see if the model behaves as expected, that is, according to specification.

First a falsifier must be registered in `lilo.toml`, e.g.

```
name = "automatic-transmission"
source = "spec"

[[system_falsifier]]
name = "AT Falsifier"
system = "transmission"
script = "transmission.py"
```

Once this is done, the falsifier will show up in the `Falsify` analysis menu. If a falsifying signal is found, the monitoring tree is shown, to help understand how the model went wrong:



## Export

`Export` converts your specifications to other formats, to be used in other tools. For example, if you want to export your specification to JSON format, choose `.json` as the `Export` type.

Analyzing spec: humidity\_correlation

Export

Run Analysis

Save Analysis

▼ Show advanced options

Export type

Record Encoding

.json

Preserve records

Record Encoding (for config)

Preserve records

System Parameters

Input.JSON

{}

```
{
  "contents": [
    "Always",
    {
      "end": null,
      "start": {
        "contents": 0,
        "tag": "Lit"
      }
    }
  ],
}
```

## Next Steps

This tour covered the basics of what SpecForge can do. The following chapters dive deeper into:

- The full Lilo language ([Lilo Language](#))
- System definitions and composition ([Systems](#))
- The Python SDK for programmatic access ([Python SDK](#))

# Lilo Language

Lilo is a formal specification language designed for describing, verifying and monitoring the behavior of complex, time-dependent systems.

Lilo allows you to:

- Write expressions using a familiar syntax with powerful temporal operators for defining properties over time.
- Define data structures using records to model your system's data.
- Structure your specifications using systems.

## Types and Expressions

Lilo is an expression-based language. This means that most constructs, from simple arithmetic to complex temporal properties, are expressions that evaluate to a time-series value. This section details the fundamental building blocks of Lilo expressions.

### Comments

Comment *blocks* start with `/*` and end with `*/`. Everything between these markers is ignored.

A *line comment* start with `//`, and indicates that the rest of the line is a comment.

*Docstrings* start with `///` instead of `//`, and attach documentation to various language elements.

### Primitive types

Lilo is a typed language. The primitive types are:

- `Bool`: Boolean values. These are written `true` and `false`.
- `Int`: Integer values, e.g. `42`.
- `Float`: Floating point values, e.g. `42.3`.
- `String`: Text strings, written between double-quotes, e.g. `"hello world"`.

Type errors are signaled to the user like this:

```
def x: Float = 1.02
def n: Int = 42
def example = x + n
```

The code blocks in this documentation page can be edited, for example try changing the type of `n` to `Float` to fix the type error.

### Operators

Lilo uses the following operators, listed in order of precedence (from highest to lowest).

- Prefix negation: `-x` is the additive inverse of `x`, and `!x` is the negation of `x`.
- Multiplication and division: `x * y`, `x / y`.
- Addition and subtraction: `x + y` and `x - y`.
- Numeric comparisons:
  - `==`: equality
  - `!=`: non-equality
  - `>=`: greater than or equals
  - `<=`: less than or equals
  - `>`: greater than (strict)
  - `<`: less than (strict) Comparisons can be chained, in a consistent direction. E.g. `0 < x <= 10` means the same thing as `0 < x && x <= 10`.
- Temporal operators
  - `always  $\phi$` :  $\phi$  is true at all times in the future.
  - `eventually  $\phi$` :  $\phi$  is true at some point in the future.



- `past  $\phi$` :  $\phi$  was true at some time in the past.
- `historically  $\phi$` :  $\phi$  was true at all times in the past.
- `will_change  $\phi$` :  $\phi$  changes value at some point in the future.
- `did_change  $\phi$` :  $\phi$  changed value at some point in the past.
- `$\phi$  since  $\psi$` :  $\phi$  is true at all points in the past, from some point where  $\psi$  was true.
- `$\phi$  until  $\psi$` :  $\phi$  is true at all points in the future until  $\psi$  becomes true.
- `next  $\phi$` :  $\phi$  is true at the *next* (discrete) time point.
- `previous  $\phi$` :  $\phi$  is true at the *previous* (discrete) time point.

Temporal operators can be qualified with intervals:

- `always [a, b]  $\phi$` :  $\phi$  is true at all times between *a* and *b* time units in the future.
  - `eventually [a, b]  $\phi$` :  $\phi$  is true at some point between *a* and *b* time units in the future.
  - `$\phi$  until [a, b]  $\psi$` :  $\phi$  is true at all points between now and some point between *a* and *b* time units in the future until  $\psi$  becomes true.
  - Similar interval qualifications apply to other temporal operators.
  - One can use `infinity` in intervals: `[0, infinity]`.
- Conjunction: `x && y`, both *x* and *y* are true.
  - Disjunction: `x || y`, one of *x* or *y* is true.
  - Implication and equivalence:
    - `x => y`: if *x* is true, then *y* must also be true.
    - `x <=> y`: *x* is true if and only *y* is true.

Note that prefix operators cannot be chained. So one must write `-(~x)`, or `!(next  $\phi$ )`.

## Built-in functions

There are built-in functions:

- `float` will produce a `Float` from an `Int`:
 

```
def n: Int = 42

def x: Float = float(n)
```
- `time` will return the current time of the signal.

## Conditional Expressions

Conditional expressions allow a specification to evaluate to different values based on a boolean condition. They use the `if - then - else` syntax.

```
if x > 0 then "positive" else "non-positive"
```

A key feature of Lilo is that `if / then / else` is an **expression**, not a statement. This means it always evaluates to a value, and thus the `else` branch is mandatory.

The expression in the `if` clause must evaluate to a `Bool`. The `then` and `else` branches must produce values of a compatible type. For example, if the `then` branch evaluates to an `Int`, the `else` branch must also evaluate to an `Int`.

Conditionals can be used anywhere an expression is expected, and can be nested to handle more complex logic.

```
// Avoid division by zero
def safe_ratio(numerator: Float, denominator: Float): Float =
  if denominator != 0.0 then
    numerator / denominator
  else
    0.0 // Return a default value

// Nested conditional
def describe_temp(temp: Float): String =
  if temp > 30.0
  then "hot"
  else if temp < 10.0
  then "cold"
  else
    "moderate"
```

Note that `if _ then _ else _` is *pointwise*, meaning that the condition applies to all points in time, independently.

## Records

Records are composite data types that group together named values, called fields. They are essential for modeling structured data within your specifications.

The Lilo language supports anonymous, structurally typed, extensible records.

### Construction and Type

You can construct a record value by providing a comma-separated list of `field = value` pairs enclosed in curly braces. The type of the record is inferred from the field names and the types of their corresponding values.

For example, the following expression creates a record with two fields: `foo` of type `Int` and `bar` of type `String`.

```
{ foo = 42, bar = "hello" }
```

The resulting value has the structural type `{ foo: Int, bar: String }`. The order of fields in a constructor does not matter.

You can also declare a named record type using a `type` declaration, which is highly recommended for clarity and reuse.

```
/// Represents a point in a 2D coordinate system.
type Point = { x: Float, y: Float }

// Construct a value of type Point
def origin: Point = { x = 0.0, y = 0.0 }
```

### Field punning

When you already have a name in scope that should be copied into a record, you can *pun* the field by omitting the explicit assignment. A pun such as `{ foo }` is shorthand for `{ foo = foo }`.

```
def foo: Int = 42
def bar: String = "hello"

def record_with_puns = { foo, bar }
```

Punning works anywhere record fields are listed, including in record literals and updates. Each pun expands to a regular `field = value` pair during typechecking.

## Path field construction

Nested records can be created or extended in one step by assigning to a dotted path. Each segment before the final field refers to an enclosing record, and the compiler will merge the pieces together.

```
type Engine = { status: { throttle: Int, fault: Bool } }

def default_engine: Engine =
  { status.throttle = 0, status.fault = false }
```

The order of path assignments does not matter; the paths are merged into the final record. A dotted path cannot be combined with punning; write `{ status.throttle = throttle }` instead of `{ status.throttle }` when you need the path form.

## Record updates with with

Use `{ base with fields }` to copy an existing record and override specific fields. Updates respect the same syntax rules as record construction: you can mix regular assignments, puns, and dotted paths.

```
type Engine = { status: { throttle: Int, fault: Bool } }

def base: Engine =
  { status.throttle = 0, status.fault = false }

def warmed_up: Engine =
  { base with status.throttle = 70 }

def acknowledged: Engine =
  { warmed_up with status.fault = false }
```

All updated fields must already exist in the base record. Path updates let you rewrite deeply nested pieces without rebuilding the entire structure.

## Projection

To access the value of a field within a record, you use the dot (`.`) syntax. If `p` is a record that has a field named `x`, then `p.x` is the expression that accesses this value.

```
type Point = { x: Float, y: Float }

def is_on_x_axis(p: Point): Bool =
  p.y == 0.0
```

Records can be nested, and projection can be chained.

```
type Point = { x: Float, y: Float }
type Circle = { center: Point, radius: Float }

def is_unit_circle_at_origin(c: Circle): Bool =
  c.center.x == 0.0 && c.center.y == 0.0 && c.radius == 1.0
```

## Local Bindings

Local bindings allow you to assign a name to an expression, which can then be used in a subsequent expression. This is accomplished using the `let` keyword and is invaluable for improving the clarity, structure, and efficiency of your specifications.

A local binding takes the form `let name = expression1; expression2`. This binds the result of `expression1` to `name`. The binding `name` is only visible within `expression2`, which is the scope of the binding.

The primary purposes of `let` bindings are:

1. **Readability:** Breaking down a complex expression into smaller, named parts makes the logic easier to follow.
2. **Re-use:** If a sub-expression is used multiple times, binding it to a name avoids repetition and potential re-computation.

Consider the following formula for calculating the area of a triangle's circumcircle from its side lengths  $a$ ,  $b$ , and  $c$ :

```
def circumcircle(a: Float, b: Float, c: Float): Float =  
  (a * b * c) / sqrt((a + b + c) * (b + c - a) * (c + a - b) * (a + b - c))
```

Using `let` bindings makes the logic much clearer:

```
def circumcircle(a: Float, b: Float, c: Float): Float =  
  let pi = 3.14;  
  let s = (a + b + c) / 2.0;  
  let area = sqrt(s * (s - a) * (s - b) * (s - c));  
  let circumradius = (a * b * c) / (4.0 * area);  
  circumradius * circumradius * pi
```

The type of the bound variable (`s`, `area`, `circumradius`) is automatically inferred from the expression it is assigned. You can also chain multiple `let` bindings to build up a computation step-by-step.

## Systems

Ultimately Lilo is used to specify *systems*. A system groups together declarations for the temporal input *signals*, the (non-temporal) *parameters* and the *specifications*. A system also includes auxiliary definitions.

A system file should start with a system declaration, e.g.:

```
system Engine
```

The name of the system should match the file name.

## Type declarations

A new type is declared with the `type` keyword. To define a new record type `Point` :

```
type Point = { x: Float, y: Float }
```

We can then use `Point` as a type anywhere else in the file.

## Signals

The time varying values of the system are called *signals*. They are declared with the `signal` keyword. E.g.:

```
signal x: Float
signal y: Float
signal speed: Float
signal rain_sensor: Bool
signal wipers_on: Bool
```

The definitions and specifications of a system can freely refer to the system's signals.

A signal can be of any type that does not contain function types, i.e. a combination of primitive types and records.

## System Parameters

Variables of a system which are constant over time are called *system parameters*. They are declared with the `param` keyword. E.g.:

```
param temp_threshold: Float
param max_errors: Int
```

The definitions and specifications of a system can freely refer to the system's parameters. Note that system parameters must be provided upfront before monitoring can begin. For exemplification, system parameters are optional. That is, they can be provided, in which case the example must conform to them, or otherwise the exemplification process will try to find values that work.

## Definitions

A definition is declared with the `def` keyword:

```
def foo: Int = 42
```

A definition can depend on parameters:

```
def foo(x: Float) = x + 42
```

One can also specify the return type of a definition:

```
def foo(x: Float): Float = x + 42
```

The type annotations on parameters and the return type are both optional, if they are not provided they are inferred. It is recommended to always specify these types as a form of documentation.

The parameters of a definition can also be record types, for instance:

```
type S = { x: Float, y: Float }  
def foo(s: S) = eventually [0,1] s.x > s.y
```

Definitions can be used in other definitions, e.g.:

```
type S = { x: Float, y: Float }  
def more_x_than_y(s: S) = s.x > s.y  
def foo(s: S) = eventually [0,1] more_x_than_y(s)
```

Definitions can be specified in any order, as long as this doesn't create any circular dependencies.

Definitions can freely use any of the signals of the system, without having to declare them as parameters.

## Specifications

A `spec` says something that should be true of the system. They can use all the `signal`s and `def`s of the system. They are declared using the `spec` keyword. They are much like `def`s except:

- The return type is always `Bool` (and doesn't need to be specified)
- They cannot have parameters.

Example:

```
signal speed: Float  
def above_min = 0 <= speed  
def below_max = speed <= 100  
spec valid_speed =  
  always (above_min && below_max)
```

## Modules

Lilo language supports modules. A module starts with a module declaration, and contains definitions (much like a system):

```
module Util

def add(x: Float, y: Float) = x + y

pub def calc(x: Float) = add(x, x)
```

A module can only contain `def`s and `type`s.

Those definitions which should be accessible from other modules should be parked as `pub`, which means "public".

To use a module, one needs to import it, e.g. `import Util`. The `pub` lic definitions from `util` are then available to be used, with qualified names, e.g.:

```
import Util

def foo(x: Float) = Util::calc(x) + 42
```

One can import a module qualified with an alias, for example:

```
import Util as U

def foo(x: Float) = U::calc(x) + 42
```

To use symbols without a qualifier, use the `use` keyword:

```
import Util use { calc }

def foo(x: Float) = calc(x) + 42
```

## Static Analysis

System code goes through some code quality checks.

### Consistency Checking

Specs are checked for consistency. A warning is produced if specs may be unsatisfiable:

```
signal x: Float
spec main = always (x > 0 && x < 0)
```

This means that the specification is problematic, because it is impossible that any system satisfies this specification.

Inconsistencies between specs are also reported to the user:

```
signal x: Float
spec always_positive = always (x > 0)
spec always_negative = always (x < 0)
```

In this case each of the specs are satisfiable on their own, but taken together they cannot be satisfied by any system.

### Redundancy Checking

If one spec is redundant, because implied by other specs of the system, this is also detected:

```
signal x: Float
spec positive_becomes_negative = always (x > 0 => eventually x < 0)
spec sometimes_positive = eventually x > 0
spec sometimes_negative = eventually x < 0
```

In this case we warn the user that the spec `sometimes_negative` is redundant, because this property is already implied by the combination of `positive_becomes_negative` and `sometimes_positive`. Indeed `sometimes_positive` implies that there is some point in time where  $x > 0$ , and using `positive_becomes_negative` we conclude that therefore there must be some point in time after than when  $x < 0$ .



# Additional Features

## Attributes

In addition to docstrings (which begin with `///`), Lilo definitions, specs, params and signals can be annotated with attributes. They must immediately precede the item they annotate.

The attributes are used to convey metadata about items they annotate which is used by the tooling (notably the VSCode extension).

```
#[key = "value", fn(arg), flag]
spec foo = true
```

- Suppressing unused variable warnings: `#[disable(unused)]`
  - Using this attribute on a definition, param or signal will suppress warnings about it being unused.
  - Specs or public definitions are always considered used.
- Timeout for Static Analyses
  - To override the default timeout for static analyses, a `timeout` can be specified in seconds.
  - They can be specified individually `#[timeout(satisfiability = 20, redundancy = 30)]`
  - Or together `#[timeout(10)]` which sets both to 10 seconds.
- Disabling static analyses
  - Use `#[disable(satisfiability)]` or `#[disable(redundancy)]` to disable specific static analyses on a definition.

## Default Values for Parameters

When specifying a system, parameters can optionally be given a default value. The intent of such a default value is to indicate that the parameter is expected to be instantiated with the default value in a typical use case.

```
param temperature: Float = 25.0
```

Default values should not be used to declare constants. For them, use a `def` instead.

```
def pi: Float = 3.14159
```

- When monitoring, parameters with default values can be omitted from the input. If omitted, the default value is used. They can also be explicitly provided, in which case the provided value is used.
- When exporting a formula, parameters with a default value will be substituted with the default value before the export.
- When exemplifying, the exemplifier will require the solver to fix the parameter to the default value.

When running an analysis such as export or exemplification, one can provide the JSON `null` value for a field in the config. This has the effect of requesting SpecForge to ignore the default value for the parameter.

```
system main

signal p: Int
param bound: Int = 1

spec foo = p > 1 && p < bound
```

- **Exemplification**

- With `params = {}` : Unsatisfiable (the default value of `bound` is used)
- With `params = { "bound": null }` : Satisfiable (the solver is free to choose a value of `bound` that satisfies the constraints)

- **Export**

- With `params = {}`, result: `p > 1 && p < 1`
- With `params = { "bound": 100 }`, result: `p > 1 && p < 100`
- With `params = { "bound": null }`, result: `p > 1 && p < bound`

Note that the JSON `null` value cannot be used as a default value as a part of the Lilo program.

## Spec Stubs

The user may create a spec stub, a spec without a body. Such a stub may still have a docstring and attributes. This can be used as a placeholder, and is interpreted as `true` by the Lilo tooling.

The VSCode extension will display a codelens to generate an implementation for the stub based on the docstring using an LLM (if configured).

```
/// The system should always eventually recover from errors.  
spec error_recovery
```

# Conventions

Some languages require that certain classes of names be capitalized or not, to distinguish them. Lilo is flexible, so that it can match the naming conventions of the system it is being used to specify. That said, here are the conventions that we use in the examples:

- Module and systems are lowercase [snake\\_case](#). So e.g. `climate_control` rather than `ClimateControl`.
- Names of `signal S`, `param S`, `def S`, `spec S`, arguments and record field names are lowercase and [snake\\_case](#). So e.g. `signal wind_speed` rather than `signal WindSpeed` or `signal windSpeed`.
- Types, including user defined ones, should be capitalized and [CamelCase](#). E.g.

```
type Plane = {  
  wind_speed: Float,  
  ground_speed: Float  
}
```

# VSCode Extension

The SpecForge VSCode extension provides a comprehensive development environment for writing and analyzing Lilo specifications. It combines language support, interactive analysis tools, and visualization capabilities in a unified interface.

See the [VSCode extension installation guide](#) to get setup.

## Overview

The extension provides:

- **Language Support:** Syntax highlighting, type-checking, and autocompletion via a Language Server Protocol (LSP) implementation
- **Diagnostics:** Real-time checking for type errors, unused definitions, and optimization suggestions
- **Code Lenses:** Interactive analysis tools embedded directly in your code
- **Spec Status Pane:** A dedicated sidebar for navigating specifications and saved analyses
- **Spec Analysis Pane:** Interactive GUI for spec monitoring, exemplification, falsification, etc.
- **Notebook Integration:** Support for visualizing SpecForge results in Jupyter notebooks
- **LLM Features:** AI-powered spec generation and diagnostic explanations

## Configuration

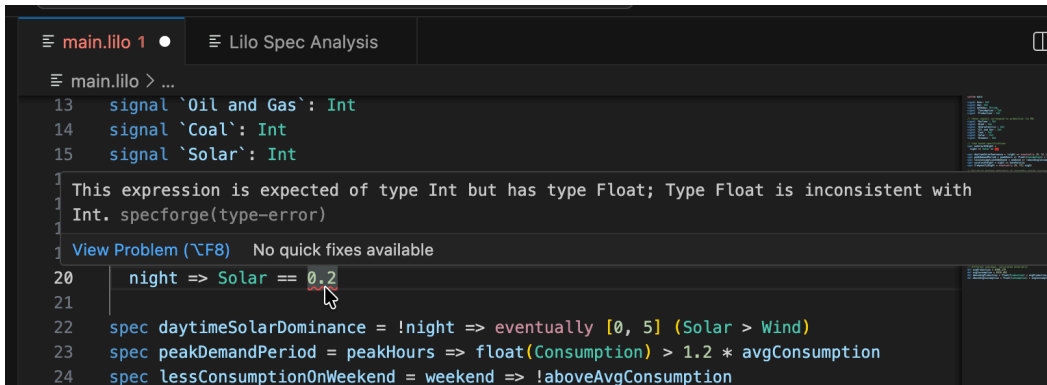
The extension requires the SpecForge server to be running. Configure the server connection in VSCode settings:

- **API Base URL:** The URI for the SpecForge server (default: `http://localhost:8080`)
  - Access via `Settings` → `Extensions` → `SpecForge` → `Api Base Url`
  - Or use the setting ID: `specforge.apiBaseUrl`
- **Enable Preview Features:** Enable experimental features including the Spec Status sidebar
  - Access via `Settings` → `Extensions` → `SpecForge` → `Enable Preview Features`
  - Or use the setting ID: `specforge.enablePreviewFeatures`

## Language Features

### Parsing and Type Checking

The extension performs real-time checking as you write specifications. Errors will be underlined. Hovering over the affected code will show the error:

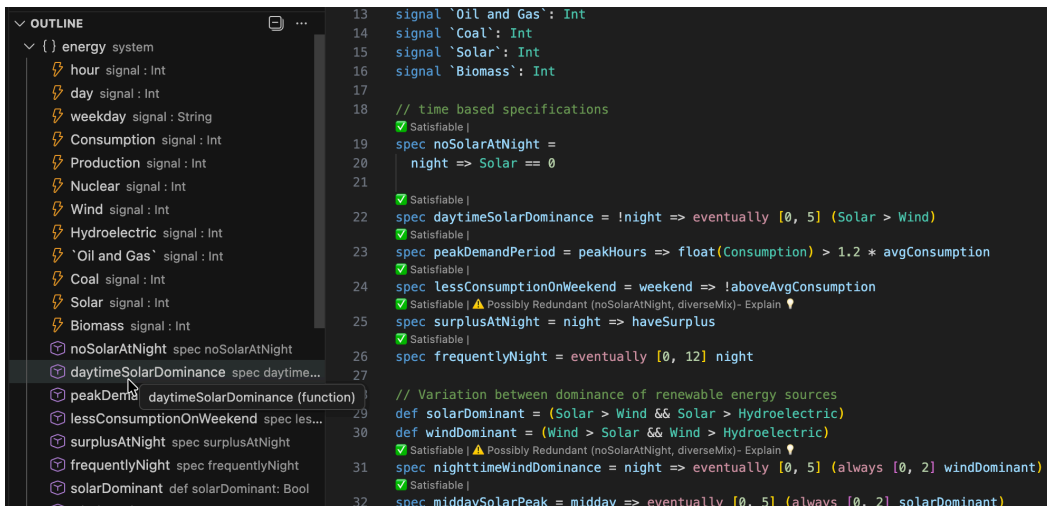


The screenshot shows the VS Code editor with a file named `main.lilo`. The code defines several signals: `Oil and Gas`, `Coal`, and `Solar`, all of type `Int`. A specification `night => Solar == 0,2` is highlighted with a red squiggly line under the `0,2`, indicating a type error. A tooltip message states: "This expression is expected of type Int but has type Float; Type Float is inconsistent with Int. specforge(type-error)". Below the error, a button says "View Problem (⌘F8) No quick fixes available". The rest of the code includes specifications for `daytimeSolarDominance`, `peakDemandPeriod`, and `lessConsumptionOnWeekend`.

The extension will check for syntax errors, type errors, etc.

## Document Outline

The extension provides a hierarchical outline of your specification file:

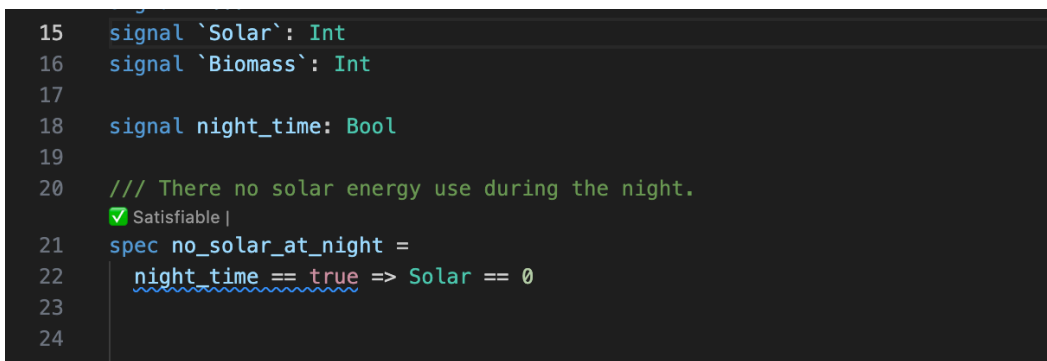


The screenshot shows the VS Code editor with the "Outline" view open in the Explorer sidebar. The outline lists the following items: `energy system`, `hour signal : Int`, `day signal : Int`, `weekday signal : String`, `Consumption signal : Int`, `Production signal : Int`, `Nuclear signal : Int`, `Wind signal : Int`, `Hydroelectric signal : Int`, `'Oil and Gas' signal : Int`, `Coal signal : Int`, `Solar signal : Int`, `Biomass signal : Int`, `noSolarAtNight spec noSolarAtNight`, `daytimeSolarDominance spec daytime...`, `peakDemand daytimeSolarDominance (function)`, `lessConsumptionOnWeekend spec les...`, `surplusAtNight spec surplusAtNight`, `frequentlyNight spec frequentlyNight`, `solarDominant def solarDominant: Bool`, and `windDominant def windDominant: Bool`. The main editor shows the corresponding code for these items, with status icons (green checkmark for satisfiable, yellow triangle for possibly redundant) next to each specification.

- Open the "Outline" view in VSCode's Explorer sidebar
- See all specs, definitions, signals, and parameters at a glance
- Click any symbol to jump to its definition
- The outline updates automatically

## Diagnostics

The extension performs various checks automatically and provides feedback.



The screenshot shows the VS Code editor with a code snippet. A diagnostic message is shown above the line `night_time == true => Solar == 0`, indicating a type error. The message is: "There no solar energy use during the night." followed by a green checkmark and the word "Satisfiable". The code defines `signal 'Solar': Int`, `signal 'Biomass': Int`, `signal night_time: Bool`, and a specification `spec no_solar_at_night = night_time == true => Solar == 0`.

Hovering over a diagnostic will reveal the message.

```

15 signal `Solar`: Int
16 signal `Biomass`: Int
17
18 signal night_time: Bool
19
20 // Consider rewriting this as: night_time specforge(optimization-suggestion)
21 sp View Problem (⌘F8) No quick fixes available
22 night_time == true => Solar == 0
23
24
25

```

Diagnostics include:

- Warnings for unused `signal S`, `param S`, `def S`, etc.
- Optimization suggestions.
- Warnings about using time-dependent expression in intervals (if so configured).

## Code Lenses

Code lenses are interactive buttons that appear above specifications in your code, offering information and possibly actions.

## Satisfiability Checking

Above each specification, you'll see a code lens indicating whether the spec is satisfiable:

```

/// During the night, wind-power should dominate.
✱ Generate with LLM
spec nighttimeWindDominance

```

Here is a spec that SpecForge has detected might be unsatisfiable:

```

def solarDominant = (Solar > Wind && Solar > Hydroelectric)

def windDominant = (Wind > Solar && Wind > Hydroelectric)

✗ Possibly Unsatisfiable - Explain ⓘ |
spec solarPeak =
  eventually [0, 5] (always [0, 2] (solarDominant && windDominant))

```

The user can ask for an explanation:

```

def solarDominant = (Solar > Wind && Solar > Hydroelectric)

def windDominant = (Wind > Solar && Wind > Hydroelectric)

✗ Possibly Unsatisfiable - Explain ⓘ |
spec solarPeak =
  eventually [0, 5] (always [0, 2] (solarDominant && windDominant))

```

If SpecForge could not decide the satisfiability, it is possible to relaunch the analysis with a longer timeout.

## Redundancy

If the system detects that a spec might be redundant, a warning is show as a code lens:

```

33
34 // During the night, wind-power should dominate.
35 ✓ Satisfiable | ⚠ Possibly Redundant (noSolarAtNight, diverseMix)- Explain
36 spec nighttimeWindDominance = night => eventually [0, 5] (always [0, 2] windDominant)

```

In this case, SpecForge is indicating that the spec is implied by the specifications `noSolarAtNight` and `diverseMix`, and is therefore not necessary. By clicking `Explain`, an explanation for the redundancy is produced:

The screenshot shows the VS Code editor with a tooltip explaining the redundancy of the `nighttimeWindDominance` specification. The tooltip text is:

```

① Because at night Solar is forced to 0 by
noSolarAtNight and diverseMix ensures Wind and
Hydroelectric are nonzero, the WindDominant
condition (Wind > Solar and Wind > Hydroelectric)
becomes entailed during the night, making
nighttimeWindDominance redundant.
Source: SpecForge

```

The background code shows several specifications, including `middaySolarPeak`, `solarToWindTransition`, and `diverseMix`.

## Spec stubs

If a `spec` does not have a body, it is a *spec stub*. In this case a code lens offers to generate the specification using AI.

```

// During the night, wind-power should dominate.
✱ Generate with LLM
spec nighttimeWindDominance

```

Clicking `Generate with LLM` will produce a definition for the specification, that works with the current system. If the spec is too ambiguous, or if there is some other obstacle to generation, an error message will be shown.

## Spec Status Pane

To access the spec status panel, click the SpecForge icon on the left hand side of VSCode:

The screenshot shows the VS Code interface with the SpecForge icon (a lightning bolt) highlighted in the left sidebar. The SpecForge status pane is open, displaying a list of specifications and their status:

- energy system
  - hour signal : Int
  - SpecForge
  - weekday signal : String
  - Consumption signal : Int
  - Production signal : Int
  - Nuclear signal : Int
  - Wind signal : Int
  - Hydroelectric signal : Int

The right pane shows the code for the `noSolarAtNight` specification:

```

// these signals correspond to
signal `Nuclear`: Int
signal `Wind`: Int
signal `Hydroelectric`: Int
signal `Oil and Gas`: Int
signal `Coal`: Int
signal `Solar`: Int
signal `Biomass`: Int

// time based specifications
✓ Satisfiable | ⚠ Possibly Redundant (solarPe
spec noSolarAtNight =
  night => Solar == 0
✓ Satisfiable | ⚠ Possibly Redundant (solarPe
spec daytimeSolarDominance = !n

```

The sidebar lists all the specification files and the specs that are defined:

The screenshot shows the SPECFORGE: EASY ANALYSIS interface. On the left, a sidebar lists specifications under the heading 'Specifications'. The list includes: `energy.lilo`, `frequentlyNight`, `daytimeSolarDominance`, `noSolarAtNight`, `diverseMix`, `nighttimeWindDominance`, `solarPeak`, `solarToWindTransition`, `lessConsumptionOnWeekend`, `demandSpikeMitigation`, `surplusAtNight`, `peakDemandPeriod`, `frequentlyMoreRenewable`, `renewableAndSurplus`, and `renewableDominanceStability`. Each item has an 'Analyse' button next to it. On the right, a code editor shows the content of `energy.lilo`. It starts with a 'Consistent system' check, followed by a `system energy` declaration. Then, several signals are defined: `hour: Int`, `weekday: String`, `Consumption: Int`, and `Production: Int`. A comment indicates these correspond to production in MW. Next, more signals are defined: `Nuclear: Int`, `Wind: Int`, `Hydroelectric: Int`, `Oil and Gas: Int`, `Coal: Int`, `Solar: Int`, and `Biomass: Int`. A section for 'time based specifications' follows, with a 'Satisfiable' check for `noSolarAtNight` (defined as `night => Solar == 0`). Another 'Satisfiable' check is shown for `daytimeSolarDominance` (defined as `!night => eventually [0`). The last line shows a 'Satisfiable' check for `peakDemandPeriod` (defined as `peakHours => float(Consumpti`).

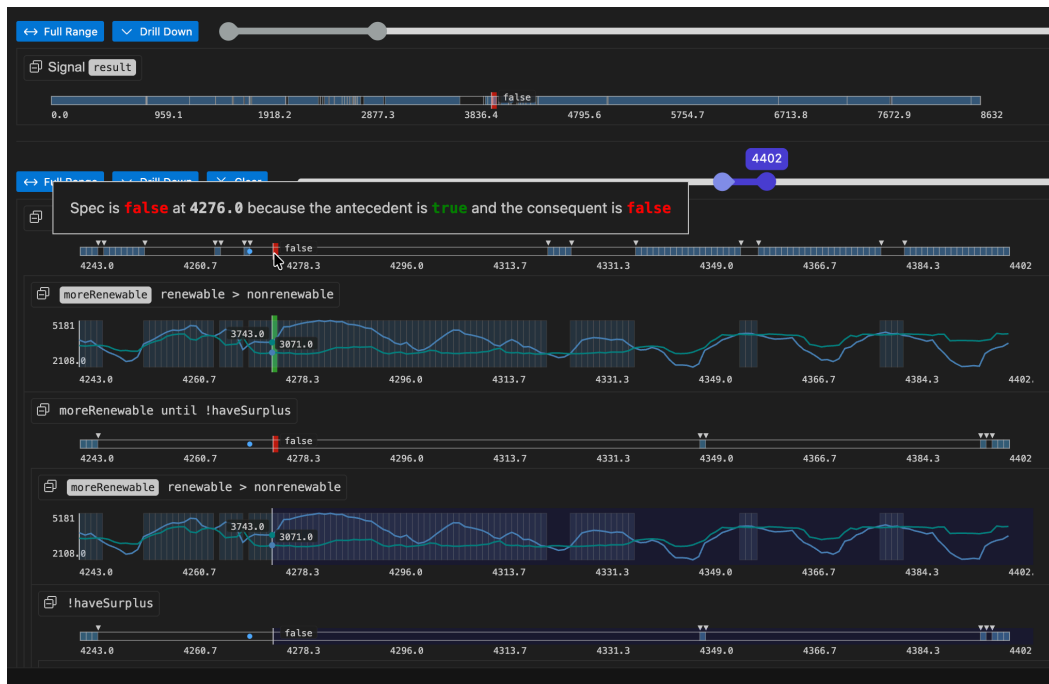
Clicking `Analyse` next to a spec will bring you to a spec analysis window. You can use this to launch various spec analysis tasks: monitoring, exemplification, export, animation and falsification.

For example, to monitor a specification, select `Monitor` from the dropdown, and choose a data file to monitor, and click `Run Analysis`.

The screenshot shows the 'Analyzing spec: renewableDominanceStability' window. At the top, there's a dropdown menu set to 'Monitor' and two buttons: 'Run Analysis' and 'Save Analysis'. Below this, there's a 'Data File' field with 'romania.csv' entered. A 'Show advanced options' button is visible. Under 'advanced options', there's a 'Full Range' button, a 'Drill Down' button, and a slider. At the bottom, there's a 'Signal' field with 'result' entered. Below the signal field is a timeline plot showing the result of the analysis. The timeline has a blue background, indicating the specification is true. The x-axis has values: 0.0, 959.1, 1918.2, 2877.3, 3836.4, 4795.6, 5754.7, and 67. The plot shows a single green vertical bar at the end, labeled 'true'.

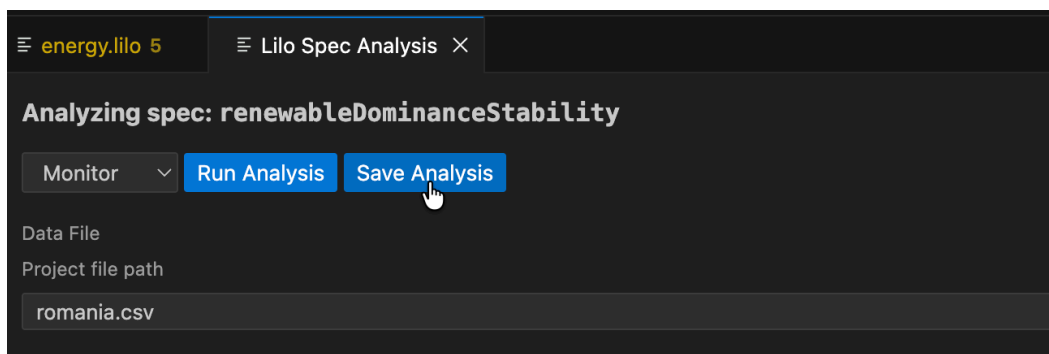
The result of the analysis is shown. The blue areas represent the times where the specification is true. For large data files, only the boolean result is shown. To better understand why a specification is false at some point, select a point and click `Drill Down`.





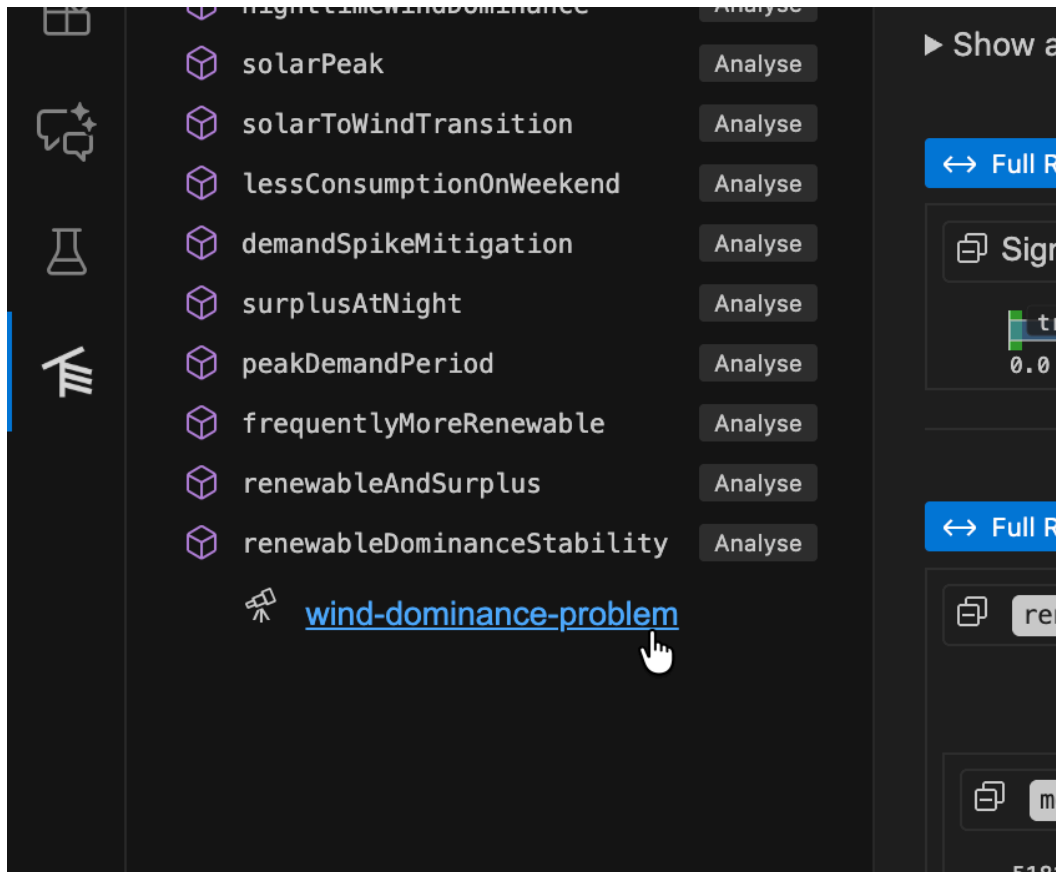
The drill-down has chosen a small enough segment of the full data source in order to present the debugging tree. This will show a tree of monitoring result for the whole specification. Each node can be collapsed or expanded. Hovering on the timeline will also highlight relevant regions in sub-expression result timelines. Hovering on a timeline will display an explanation of why the result is true or false at that point in time, for that sub-expression.

A spec analysis such as this can be *saved* by clicking on the `Save Analysis` button.



Choose a location in your project to save the analysis.

Saved analyses will show up in the spec status side panel, underneath the relevant spec.



## Analysis Types

The GUI supports five types of analysis:

### 1. Monitor

Check whether recorded system behavior satisfies your specification.

#### Inputs:

- **Signal Data:** CSV, JSON, or JSONL file containing time-series data
  - CSV: Column headers must match signal names
  - JSON/JSONL: Objects with keys matching signal names
- **Parameters:** JSON object with parameter values
- **Options:** Monitoring configuration (see [Monitoring Options](#))

#### Output:

- Monitoring tree showing spec satisfaction over time
- Drill-down into sub-expressions
- Visualization of signal values

#### Example:

```
{  
  "min_temperature": 10.0,  
  "max_temperature": 30.0  
}
```

### 2. Exemplify

Generate example traces that satisfy your specification.

#### Inputs:

- **Number of Points:** How many time points to generate (default: 10)
- **Timeout:** Maximum time to spend generating (default: 5 seconds)
- **Also Monitor:** Whether to also show monitoring tree for the generated trace
- **Assumptions:** Additional constraints to satisfy (array of spec expressions)

#### Output:

- Generated signal data as time-series
- Optional monitoring tree if "Also Monitor" is enabled
- CSV/JSON export of generated data

#### Use Cases:

- Understanding what valid behavior looks like
- Testing other components with realistic data
- Creating test fixtures
- Validating your specification makes sense

### 3. Falsify

Search for counterexamples that violate your specification using an external model.

#### Prerequisites:

- A falsification script must be registered in `lilo.toml`:

```
[[system_falsifier]]
name = "Temperature Model"
system = "temperature_control"
script = "falsifiers/temperature.py"
```

#### Falsification Script Protocol:

Your script receives these command-line arguments:

- `--system`: The system name
- `--spec`: The specification name
- `--options`: JSON string with options
- `--params`: JSON string with parameter values
- `--project-dir`: Path to the project root

The script should:

1. Simulate the system according to the specification
2. Search for a trace that violates the spec
3. Output JSON in the correct format with either success or failure.

#### Inputs:

- **Falsifier:** Select from configured falsifiers (dropdown)
- **Timeout:** Maximum time for falsification (default: 240 seconds)
- **Parameters:** JSON object with parameter values

#### Output:

- If counterexample found:
  1. Falsification result showing the failing trace
  2. Automatic monitoring of the counterexample
  3. Visualization of where/how the spec fails
- If no counterexample found: Success message

Make sure your script is executable:

```
chmod +x falsifiers/temperature.py
```

## 4. Export

Convert your specification to other formats.

### Export Formats:

- **Lilo:** Export as `.lilo` format with optional transformations
- **JSON:** Machine-readable JSON representation

### Inputs:

- **Export Type:** Select the target format
- **Parameters:** Parameter values (if needed for export)

### Output:

- Exported specification in the selected format
- Can be saved to a file

### Use Cases:

- Integrating with other tools
- Documentation generation
- Archiving specifications

## 5. Animate

Create animations showing specification behavior over time.

### Inputs:

- **SVG Template:** Path to SVG file with placeholders
- **Signal Data:** CSV, JSON, or JSONL file with time-series data

### Output:

- Frame-by-frame SVG images showing system evolution
- Can be combined into an animated visualization

### SVG Template Format:

Your SVG template should include `data-` attributes for signal values, e.g.:

```
<svg xmlns="http://www.w3.org/2000/svg" width="100" height="100" viewBox="0 0 40 40"
role="img" aria-label="Transformed ball">
  <rect width="100%" height="100%" fill="white"/>
  <g transform="translate(0,50) scale(1,-1)">
    <circle cx="20" data-cy="temperature" cy="0" r="3" fill="black" stroke="white"/>
  </g>
</svg>
```

In this example, the `<circle>` elements `cy` attribute will be animated by the value of the `temperature` signal, thanks to the `data-cy="temperature"` attribute.

## Monitoring Options

When running Monitor or Falsify analyses, you can configure these options:

- **Time Bounds:** Restrict monitoring to a specific time range
- **Sampling:** Adjust temporal resolution
- **Signal Filtering:** Monitor only specific signals
- (Additional options may be available)

## Working with Results

### Monitoring Tree

The monitoring tree shows:

- **Root:** Overall spec result (true/false/unknown)
- **Sub-expressions:** Drill down into why the spec is true or false
- **Timeline:** Hover over any expression to see when it's true/false
- **Highlighting:** Relevant segments are highlighted when hovering

 Monitoring Tree

### Loading Saved Analyses

Open a saved `.analysis.sf` file to:

- See the original configuration
- Re-run the analysis with the same settings
- Modify parameters and run again
- Export results

The Analysis Editor provides the same interface as the main analysis GUI, but pre-populated with your saved configuration.

An analysis is a file, if you modify the analysis, you should save it.

## Jupyter Notebook Integration

The extension includes a notebook renderer for displaying SpecForge results in Jupyter notebooks.

### Activation

The renderer automatically activates for:

- Jupyter notebooks (`.ipynb` files)
- VSCode Interactive Python windows

### Usage with Python SDK

When using the SpecForge Python SDK, results are automatically rendered:

```
from specforge import SpecForge

sf = SpecForge()
result = sf.monitor(
    spec_file="temperature_control.lilo",
    definition="always_in_bounds",
    data="sensor_logs.csv",
    params={"min_temperature": 10.0, "max_temperature": 30.0}
)

# result is automatically rendered in the notebook
result
```

## Snippets

The extension provides Python code snippets for common SpecForge operations ( `monitor` , `exemplify` , `export` ). Type the snippet name and press Tab to insert and navigate through placeholders.

## Troubleshooting

### Extension Not Working

#### Check the SpecForge server:

- Ensure the server is running (see [SpecForge Server](#))
- Verify the API base URL in settings matches your server
- Check the server logs for errors

#### Restart the language server:

- Run `SpecForge: Restart Language Server` from the command palette
- Check the "Output" panel (View → Output) and select "SpecForge" from the dropdown

### Diagnostics Not Appearing

#### Trigger a refresh:

- Save the file (Ctrl+S / Cmd+S)
- Close and reopen the file
- Restart the language server

#### Check server connection:

- Look at the status bar for connection status
- Verify the server is reachable at the configured URL

### Code Lenses Not Showing

#### Check configuration:

- Ensure code lenses are enabled in VSCode: `Editor > Code Lens`
- Save the file to trigger code lens computation

#### Check for errors:

- Look for parse or type errors that prevent analysis
- Fix any red squiggles in your code

### Analysis GUI Not Loading

#### Check webview:

- Open the Developer Tools: `Help > Toggle Developer Tools`
- Look for errors in the Console tab
- Check if the webview iframe loaded

#### Check server connection:

- Verify the API base URL is correct
- Test the URL in a browser (should show a JSON response)

## Falsification Script Errors

### Verify script setup:

- Check the script path in `lilo.toml`
- Ensure the script is executable: `chmod +x script.py`
- Test the script manually with example arguments

### Check script output:

- The script must output valid JSON
- Use the correct result format (see [Falsify](#))
- Check script logs for error messages

### Common issues:

- `ENOENT` : Script file not found (check path)
- `EACCES` : Script not executable (run `chmod +x` )
- Parse error: Invalid JSON output (check script output format)

# SpecForge Python SDK

The SpecForge python SDK is used for interacting with the SpecForge API, enabling formal specification monitoring, animation, export, and exemplification.

Refer to the [Setting up the Python SDK](#) guide for instructions on installing and configuring the SDK in a Python environment.

## Quick Start

```
from specforge_sdk import SpecForgeClient

# Initialize client
client = SpecForgeClient(base_url="http://localhost:8080")

# Check API health
if client.health_check():
    print("✓ Connected to SpecForge API")
    print(f"API Version: {client.version()}")
else:
    print("✗ Cannot connect to SpecForge API")
```

## Core Features

The SDK provides access to core SpecForge capabilities:

- **Monitoring:** Check specifications against data
- **Animation:** Create visualizations over time
- **Export:** Convert specifications to different formats
- **Exemplification:** Generate example data that satisfies specifications

## Documentation

See the comprehensive demo notebook at `sample-project/demo.ipynb` for:

- Detailed usage examples
- Jupyter notebook integration
- Custom rendering features
- Error handling patterns
- Complete API reference

## API Methods

- `monitor(spec_file, definition, data_file, ...)` - Monitor specifications
- `animate(spec_file, data_file, svg_file, ...)` - Create animations
- `export(spec_file, definition, export_type, ...)` - Export specifications
- `exemplify(spec_file, definition, n_points, ...)` - Generate examples
- `health_check()` - Check API availability
- `version()` - Get API version



## File Format Support

- **Specifications:** `.lilo` files
- **Data:** `.csv`, `.json`, `.jsonl` files
- **Visualizations:** `.svg` files

## Requirements

- Python 3.12+
- `requests`  $\geq 2.25.0$
- `urllib3`  $\geq 1.26.0$

# Temperature Sensor - SpecForge Sample Project

This is a complete example demonstrating the SpecForge Python SDK capabilities.

## Files

- `temperature_monitoring.lilo` - Sample specification for temperature monitoring
- `sensor_data.csv` - Sample sensor data (31 data points with temperature and humidity)
- `demo.ipynb` - Jupyter notebook demonstrating all SDK features
- `temperature_config.json` - Configuration file (parameters) for the temperature monitoring example
- `util.lilo` - Utility functions for the example

## Setup

---

[!NOTE] Hereafter, we recommend users to open `temperature_sensor` folder with VSCode.

---

### 1. Create and Activate a Virtual Environment

It is generally recommended (but not mandatory) to do this in a virtual environment. To do so, follow these instructions:

```
# Navigate to the sample project directory
cd temperature_sensor

# Create a virtual environment
python -m venv .venv

# Activate the virtual environment
# On Windows:
.venv\Scripts\activate
# On macOS/Linux:
source .venv/bin/activate
```

### 2. Install Dependencies

```
# Install the SpecForge SDK Wheel
pip install ../specforge_sdk-xxx.whl

# Install additional dependencies for the sample project
pip install jupyter pandas matplotlib numpy
```

### 3. Verify Installation

```
# Check that the SDK is installed correctly
python -c "from specforge_sdk import SpecForgeClient; print('/ SDK installed successfully')"
```

# Running the Examples

## Prerequisites

1. Ensure SpecForge API server is running on `http://localhost:8080/health`
2. Activate your virtual environment (if not already active):

```
# On Windows:  
venv\Scripts\activate  
# On macOS/Linux:  
source venv/bin/activate
```

## Run the Examples

To check that the extension is working, execute the cells in `demo.ipynb`. The output of the monitoring commands should have a visualization.

You may need to make sure your notebook is connected to the correct Python kernel. Often, it is `.venv (Python3.X.X)` or `ipykernel`. It can be configured from the VSCode notebook interface.

## Sample Data Overview

The included `sensor_data.csv` contains:

- 31 time points (0.0 to 30.0)
- Temperature readings (20.8°C to 25.0°C)
- Humidity readings (40.9% to 51.5%)

This data is designed to test various specifications including temperature bounds, stability, and humidity correlation.

## Deactivating the Environment

When you're done working with the sample project:

```
deactivate
```

## Troubleshooting

### Common Issues

1. **"Module not found" errors:** Ensure the virtual environment is activated and the SDK is installed with `pip install ../specforge_sdk-xxx.whl`
2. **Connection refused:** Make sure the SpecForge API server is running on `http://localhost:8080/health`
3. **Jupyter not found:** Install it with `pip install jupyter` in your activated virtual environment
4. **Missing sample files:** Ensure you're in the `temperature_sensor` directory

## Verify Setup

```
# Check Python environment
which python
pip list | grep specforge

# Test API connection
python -c "from specforge_sdk import SpecForgeClient; client = SpecForgeClient();
print('Health check:', client.health_check())"
```

# Command Line Interface

The core features of the SpecForge tool can be accessed via the command line interface (CLI).

Running the SpecForge CLI without any additional arguments will display a list of available commands. Here is a brief overview of the commands.

- `serve` : Start the SpecForge server for interactive development via the VSCode extension and the Python SDK.
- `init` : Setup a new Lilo project with a sample `lilo.toml` file and a sample spec.
- `parse` : Check if the given list of Lilo files parse correctly.
- `check` : Typecheck the given list of Lilo files.
- `format` : Format the given list of Lilo files, possibly inlining specs.
- `export` : Export a Lilo spec to another formalism (e.g., `.json` or `.lilo`).
- `eval` : Evaluate a Lilo spec on a given input signal, i.e, determine the value of the spec at the first timestamp of the input signal.
- `monitor` : Monitor a Lilo spec on a given input signal, i.e., produce an output signal by evaluating the spec at every timestamp of the input signal.
- `streammon` : Monitor a Lilo spec in streaming mode. This mode will read the input signal line-by-line from standard input as JSONL records, and output the value of the spec (when applicable) in a streaming manner.

---

The documentation for each CLI tool should be accessed directly using the `--help` flag with commands. Here is an example:

```
$ cabal exec reqeng -- monitor --help
Usage: reqeng monitor PROJDIR SYSTEM DATAFILE PARAMFILE DEFNAME
        [--file-format FORMAT] [RECORD_ENCODING]

    Monitor a spec file

Available options:
  PROJDIR      Path to the project directory
  SYSTEM       System in which the spec lives
  DATAFILE    Path to the data file
  PARAMFILE    Path to the param file
  DEFNAME      Name of the definition to monitor
  --file-format FORMAT  One of these formats: csv, json, jsonl
  -h,--help    Show this help text

Record Encodings:
  flat      Flat JSON with separator
  nested    Nested JSON
```

# Changelog

All notable changes will be documented here. The format is based on [Keep a Changelog](#). Lilo adheres to [Semantic Versioning](#).

## Unreleased

### v0.5.1 - 2025-11-05

#### Added

- New documentation site: <https://docs.imiron.io/>.
- You can now create animation gif animations.
- Projects are setup with a `lilo.toml` file, see [Project Configuration](#).
- Registration of system falsifiers in `lilo.toml`.
- VSCode spec status now lists analysis.
- Spec analysis in VSCode.
- Run falsification engines from the spec analysis pane.

#### Changed

- Better type errors for conflicting record construction/update.
- LLM explanations are localised according to user's VSCode settings.
- Unbound variable errors now include a list of in-scope variables with similar spellings.
- System globals ( `signals` and `params` ) can have attributes, including docstrings.
- The command JSON format has changed significantly, as is expected to be stable (backwards compatible) going forwards. In particular this uses system names, not filenames.
- One can specify a param as `null` (JSON) to *remove* the default param values.
- LLM spec generation will fail for under-specified specifications.
- CLI interface is updated to work with modules.

### v0.5.0 - 2025-10-14

#### Added

- Default `param S`: `param foo: Float = 42` sets `42` as the *default* value of parameter `foo`.
- Timeout attributes:

```
#[timeout = 3]
spec foo = ...
```

will set the timeout to 3 seconds for analysis tasks on `spec foo`.

- Warning for mismatched server/client versions.
- Spec stubs:

```
spec no_overheat
```

creates a "spec stub" (an unimplemented spec). There is also a code action to suggest an implementation using the docstring, using AI.

- Retry analysis with longer timeout: if an analysis times out, there is a code action to retry with a longer timeout.
- Record features:
  - Record update (including deep)
  - Field punning.
  - Path construction and path update.
- Warnings for unused `def s`, `signal s` and `param s`.
- Code hierarchy in VSCode.
- Modules: User can create modules (containing only `def` and `type` declarations), and import them.

## Changed

- VSCode code lenses resolve one at a time, which results in a much more responsive experience.