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.

A version of this guide in Japanese is also available. (□□□□□□□□□□□□□□□□□)

Getting Started

To start creating Lilo specification files, you need two things:

- The SpecForge server (via docker)
- The VSCode extension

SpecForge Server (via Docker)

Obtaining the Docker Image

Setup (One-time)

1. Save the provided JSON key somewhere, e.g.: `some_dir/key.json` .
2. Run this command at the working directory to authenticate with Google Artifact Registry:

```
cat key.json | docker login -u _json_key --password-stdin https://asia-northeast1-
docker.pkg.dev
```

Docker will now use this key to pull from Imiron's docker image registry.

Updating the Image (upon new release)

To pull the latest image, run:

```
docker compose pull
```

Configuration

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

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

For using LLM-based features such as natural-language based spec generation and error explanation, you need an OpenAI API key (which can be obtained from the OpenAI website).

To use an API key, set the environment variable `OPENAI_KEY` in your shell or replace the following line with the actual key in the `docker-compose.yml` file:

```
- OPENAI_KEY=${OPENAI_KEY}
```

▯ Without an OpenAI API key, LLM-based SpecForge features would be unavailable.

Running the Server

To run SpecForge you just need Docker Compose, which is most likely already available if you installed Docker itself. With `docker-compose.yml` in current directory run:

```
docker compose up --abort-on-container-exit
```

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

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

Lilo Language Extension for VSCode

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 extension, locate the `lilo-language-x.x.x.vsix` file, and install it in one of the following ways:

- Open the extensions tab (Ctrl+Shift+X or Cmd+Shift+X), click on the three dots at the top right, and select `Install from VSIX...`
- Open the 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.

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

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`. It can be installed directly using `pip`, or defined as a dependency via a build envionment such as `poetry` or `uv`.

See the documentation on the Python SDK for more details.

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: `""`
    - `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:

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

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

[diagnostics.consistency]
enabled = true

[diagnostics.consistency.timeouts]
named = 0.5
system = 1.0

[diagnostics.optimize]
enabled = true

[diagnostics.redundancy]
enabled = true

[diagnostics.redundancy.timeouts]
named = 0.5
system = 1.0

[diagnostics.unused_defs]
enabled = true
```

Example `lilo.toml`

An example project with overrides.

```toml
[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 falisifier"
system = "lane_keeping"
script = "falsifiers/alks.py"
```

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 φ`: `φ` is true at all times in the future.
  - `eventually φ`: `φ` is true at some point in the future.

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

  Temporal operators can be qualified with intervals:

    - `always [a, b] ϕ`: `ϕ` is true at all times between `a` and `b` time units in the future.
    - `eventually [a, b] ϕ`: `ϕ` is true at some point between `a` and `b` time units in the future.
    - `ϕ until [a, b] ψ`: `ϕ` is true at all points between now and some point between `a` and `b` time units in the future until `ψ` 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 `y` or `y` is true.
- Impliciation 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 ϕ)`.

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 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 though 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 `positivie_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 mach 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
}
```

SpecForge Python SDK

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

Installation

From Source

```
cd tools/python
pip install -e .
```

From the Wheel File

Locate the wheel file with the name `specforge_sdk-x.x.x-py3-none-any.whl` and install it using pip:

```
pip install path/to/specforge_sdk-0.5.0-py3-none-any.whl
```

Quick Start

```python
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>=2.25.0`
- `urllib3>=1.26.0`

SpecForge SDK 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 `sample_project` 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 sample_project

# 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 Pyhton 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 `sample_project` directory

Verify Setup

```bash
# 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())"
```

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 spec export format: RTAMT.
- 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 (`signal` s and `param` s) 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.