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. A few example projects can be found in the examples repository or be downloaded 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.)

You are reading the docs for version **0.5.9** of SpecForge. Navigate to a different URL to view docs for a different version.

Setting up SpecForge

The SpecForge suite consists of a few components:

- The **SpecForge Server** which is the backend server which the other components connect to. It can be run via Docker or as an executable.
- 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.

Quick Start

Follow these steps to get started quickly:

1. Install dependencies `z3` and `rsvg-converter` (see OS-specific instructions; `rsvg-converter` is optional)
2. Download and extract the SpecForge executable for your operating system
3. Configure your license (place `license.json` in the appropriate location for your OS)
4. (Optional) Configure LLM provider by setting environment variables (e.g., `SPECFORGE_LLM_PROVIDER=openai`, `OPENAI_API_KEY=...`) - see LLM Provider Configuration
5. Start the SpecForge server: `./specforge serve` (or `.\specforge.exe serve` on Windows)
6. Install the VSCode Extension (see docs)
7. Create a directory for your project and place your `.lilo` files directly in it
8. Open the directory in VSCode and start writing specifications

Note: The `lilo.toml` project configuration file is optional. For initial setup, you can skip it and place your specification and data files directly in the project root. See Project Configuration for details on when and how to use `lilo.toml`.

Detailed Setup Instructions

Choose your platform for detailed setup instructions:

- **Windows** - Complete setup guide for Windows
- **macOS** - Complete setup guide for macOS (Apple Silicon)
- **Linux** - Complete setup guide for Linux
- **Docker** - Using Docker instead of the executable

Server Environment Variables

The SpecForge server reads the following environment variables at startup. These can be set before running `specforge serve` or configured in your Docker Compose file.

Variable	Type	Default	Description
`SPECFORGE_PORT`	Integer	`8080`	Port the server listens on.
`SPECFORGE_CACHE_BOUND`	Integer	`64`	Maximum number of entries in the exemplification cache. The cache stores results from the SMT solver (Z3) to avoid redundant computations. Set to `0` to disable caching.

Variable	Type	Default	Description
`SPECFORGE_SEMAPHORE_LIMIT`	Integer	`8`	Maximum number of concurrent SMT solver (Z3) instances. Limits parallelism to prevent resource exhaustion. Set to `0` to disable the semaphore (no concurrency limit).

Example usage:

```
SPECFORGE_PORT=9090 SPECFORGE_CACHE_BOUND=128 ./specforge serve
```

For LLM-related environment variables (`SPECFORGE_LLM_PROVIDER` , `SPECFORGE_LLM_MODEL` , `OPENAI_API_KEY` , etc.), see the LLM Provider Configuration guide.

Note that all of these can also be configured in the settings of the SpecForge VSCode extension.

Setting up SpecForge on Windows

This guide will walk you through setting up SpecForge on Windows.

1. Installation

MSI Installer (Recommended)

Download `specforge-x.y.z-Windows-X64-en-US.msi` from the SpecForge releases page and run the installer.

The MSI installer will:

- Install SpecForge to `C:\Program Files\Imiron\SpecForge\`
- Add SpecForge to your system PATH automatically
- Include all required dependencies (Z3, rsvg-convert)

Standalone Executable

Download `specforge-x.y.z-Windows-X64.zip` from the SpecForge releases page and extract it to a directory of your choice.

> **Note:** With this method, you need to manually install dependencies.

The SpecForge executable requires **Z3** and **rsvg-converter** (optional) to be installed on your system.

Using Chocolatey (Recommended)

Open PowerShell as Administrator and run:

```
choco install z3 rsvg-convert
```

If you don't have Chocolatey, you can install it from chocolatey.org.

Manual Installation

If you prefer not to use a package manager, download Z3 directly from the Z3 releases page and add it to your PATH.

2. Configure Your License

The SpecForge server requires a valid license file to start. If you don't have a license, please contact the SpecForge team or request a trial license.

Place your `license.json` file in one of the following locations:

- **Standard Configuration Directory** (recommended):
  - `%APPDATA%\specforge\license.json`
  - Typically: `C:\Users\YourUsername\AppData\Roaming\specforge\license.json`
- **Environment Variable** (for custom locations):

```
$env:SPECFORGE_LICENSE_FILE="C:\path\to\license.json"
```

- **Current Directory**: `.\license.json`

Create the directory if it doesn't exist. You can do this in PowerShell:

```
New-Item -ItemType Directory -Force -Path "$env:APPDATA\specforge"
Copy-Item "C:\path\to\your\license.json" "$env:APPDATA\specforge\license.json"
```

3. Configure LLM Provider (Optional)

To use LLM-based features such as natural-language spec generation and error explanation, configure an LLM provider by setting environment variables before starting the server.

For OpenAI (recommended):

```
$env:SPECFORGE_LLM_PROVIDER="openai"
$env:SPECFORGE_LLM_MODEL="gpt-5-nano-2025-08-07"
$env:OPENAI_API_KEY="your-api-key-here"
```

Get an API key from platform.openai.com/api-keys.

For other providers (Gemini, Anthropic, Ollama), see the LLM Provider Configuration guide.

4. Start the Server

If you used the MSI installer, run from any directory:

```
specforge serve
```

If you used the standalone executable, navigate to the directory where you extracted the SpecForge executable and run:

```
.\specforge.exe serve
```

The server will start on `http://localhost:8080`. You can verify it's running by navigating to `http://localhost:8080/health`, which should show version information.

> **Note**: The server will exit immediately if the license is missing or invalid. If you encounter startup issues, verify your license configuration.

5. Install the VSCode Extension

Install the SpecForge VSCode extension from the Visual Studio Marketplace or see the VSCode Extension setup guide.

6. Install the Python SDK (Optional)

The Python SDK enables interaction with the SpecForge server programmatically from Python. This can be used to embed SpecForge analyses in Python notebooks and directly feed and retrieve data using Pandas Dataframes. See the Python SDK setup guide for instructions on how to set it up.

Further Reading

- VSCode Extension - Learn about the VSCode extension features
- Python SDK - Set up the Python SDK for programmatic access
- A Whirlwind Tour - Take a tour of SpecForge capabilities
- Project Configuration - Learn about `lilo.toml` configuration

Setting up SpecForge on macOS

This guide will walk you through setting up SpecForge on macOS using the standalone executable.

1. Download the Executable

Download `specforge-x.y.z-macOS-ARM64.tar.bz2` from the SpecForge releases page and extract it to a directory of your choice.

2. Install Dependencies

The SpecForge executable requires **Z3** to be installed. **rsvg-converter** is optional but recommended for SVG-to-PNG conversion used by the animation feature.

Install using Homebrew:

```
brew install z3 librsvg
```

If you don't have Homebrew, install it from brew.sh.

> **Note:** `librsvg` provides the `rsvg-convert` command. Without it, the `animate()` feature in the Python SDK will not be able to render PNG frames from SVG visualizations. All other SpecForge features work without it.

3. Configure Your License

The SpecForge server requires a valid license file to start. If you don't have a license, please contact the SpecForge team or request a trial license.

Place your `license.json` file in one of the following locations:

- **Standard Configuration Directory** (recommended):
  - `~/.config/specforge/license.json`
- **Environment Variable** (for custom locations):

  ```
  export SPECFORGE_LICENSE_FILE=/path/to/license.json
  ```

- **Current Directory**: `./license.json`

Create the directory if it doesn't exist:

```
mkdir -p ~/.config/specforge
cp /path/to/your/license.json ~/.config/specforge/
```

4. Configure LLM Provider (Optional)

To use LLM-based features such as natural-language spec generation and error explanation, configure an LLM provider by setting environment variables before starting the server.

For OpenAI (recommended):

```
export SPECFORGE_LLM_PROVIDER=openai
export SPECFORGE_LLM_MODEL=gpt-5-nano-2025-08-07
export OPENAI_API_KEY=your-api-key-here
```

Get an API key from platform.openai.com/api-keys.

For other providers (Gemini, Anthropic, Ollama), see the LLM Provider Configuration guide.

5. Start the Server

Navigate to the directory where you extracted the SpecForge executable and run:

```
./specforge serve
```

The server will start on `http://localhost:8080`. You can verify it's running by navigating to `http://localhost:8080/health`, which should show version information.

> **Note**: The server will exit immediately if the license is missing or invalid. If you encounter startup issues, verify your license configuration.

Allowing Execution of the Downloaded SpecForge Binary

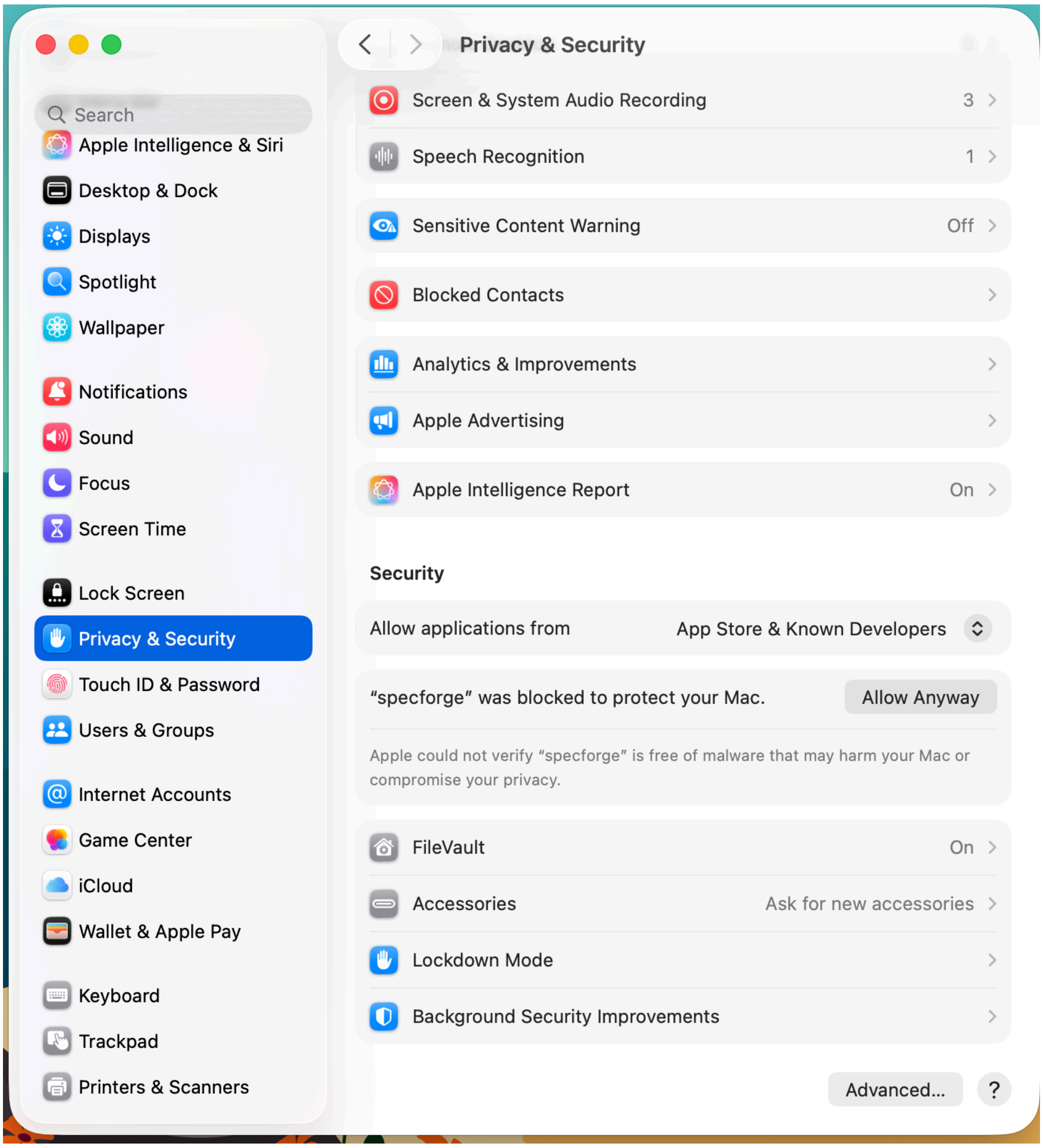
The MacOS Gatekeeper may display an alert preventing you from executing the downloaded binary, because it was downloaded from a third-party source.

To whitelist the specforge executable, run the following command.

```
xattr -d com.apple.quarantine path/to/specforge
```

Alternatively, you can do so from the System Settings GUI by following these steps.

1. Open System Settings, and go to 'Privacy & Security'
2. In the security section, you should see '"specforge" was blocked to protect your Mac.'
3. Click 'Open Anyway'.



6. Install the VSCode Extension

Install the SpecForge VSCode extension from the Visual Studio Marketplace or see the VSCode Extension setup guide.

7. Install the Python SDK (Optional)

The Python SDK enables interaction with the SpecForge server programmatically from Python. This can be used to embed SpecForge analyses in Python notebooks and directly feed and retrieve data using Pandas Dataframes. See the Python SDK setup guide for instructions on how to set it up.

Further Reading

- VSCode Extension - Learn about the VSCode extension features
- Python SDK - Set up the Python SDK for programmatic access
- A Whirlwind Tour - Take a tour of SpecForge capabilities
- Project Configuration - Learn about `lilo.toml` configuration

Setting up SpecForge on Linux

This guide will walk you through setting up SpecForge on Linux using the standalone executable.

1. Download the Executable

Download `specforge-x.y.z-Linux-X64.tar.bz2` from the SpecForge releases page and extract it to a directory of your choice.

2. Install Dependencies

The SpecForge executable requires **Z3** to be installed. **rsvg-converter** is optional but recommended for SVG-to-PNG conversion used by the animation feature.

Use your distribution's package manager:

Ubuntu/Debian:

```
sudo apt install z3 librsvg2-bin
```

Fedora/RHEL:

```
sudo dnf install z3 librsvg2-tools
```

Arch Linux:

```
sudo pacman -S z3 librsvg
```

> **Note:** The `librsvg2-bin` / `librsvg2-tools` / `librsvg` package provides the `rsvg-convert` command. Without it, the `animate()` feature in the Python SDK will not be able to render PNG frames from SVG visualizations. All other SpecForge features work without it.

3. Configure Your License

The SpecForge server requires a valid license file to start. If you don't have a license, please contact the SpecForge team or request a trial license.

Place your `license.json` file in one of the following locations:

- **Standard Configuration Directory** (recommended):
  - `~/.config/specforge/license.json`
- **Environment Variable** (for custom locations):

  ```
  export SPECFORGE_LICENSE_FILE=/path/to/license.json
  ```

- **Current Directory**: `./license.json`

Create the directory if it doesn't exist:

```
mkdir -p ~/.config/specforge
cp /path/to/your/license.json ~/.config/specforge/
```

4. Configure LLM Provider (Optional)

To use LLM-based features such as natural-language spec generation and error explanation, configure an LLM provider by setting environment variables before starting the server.

For OpenAI (recommended):

```
export SPECFORGE_LLM_PROVIDER=openai
export SPECFORGE_LLM_MODEL=gpt-5-nano-2025-08-07
export OPENAI_API_KEY=your-api-key-here
```

Get an API key from platform.openai.com/api-keys.

For other providers (Gemini, Anthropic, Ollama), see the LLM Provider Configuration guide.

5. Start the Server

Navigate to the directory where you extracted the SpecForge executable and run:

```
./specforge serve
```

The server will start on `http://localhost:8080`. You can verify it's running by navigating to `http://localhost:8080/health`, which should show version information.

> **Note**: The server will exit immediately if the license is missing or invalid. If you encounter startup issues, verify your license configuration.

6. Install the VSCode Extension

Install the SpecForge VSCode extension from the Visual Studio Marketplace or see the VSCode Extension setup guide.

7. Install the Python SDK (Optional)

The Python SDK enables interaction with the SpecForge server programmatically from Python. This can be used to embed SpecForge analyses in Python notebooks and directly feed and retrieve data using Pandas Dataframes. See the Python SDK setup guide for instructions on how to set it up.

Further Reading

- VSCode Extension - Learn about the VSCode extension features
- Python SDK - Set up the Python SDK for programmatic access
- A Whirlwind Tour - Take a tour of SpecForge capabilities
- Project Configuration - Learn about `lilo.toml` configuration

Setting up SpecForge with Docker

This guide will walk you through setting up SpecForge using Docker instead of the standalone executable.

Prerequisites

Before starting the server, you must obtain and configure a valid license. If you don't have a license, please contact the SpecForge team or request a trial license.

1. Obtain the Docker Compose File

The SpecForge Server is distributed as a Docker Image via GHCR (GitHub Container Registry). The recommended way to run the Docker Image is through Docker Compose.

Download the latest `docker-compose-x.y.z.yml` file from the SpecForge releases page.

2. Configure Your License

The SpecForge server requires a valid license file. You need to make the license file available to the Docker container.

1. Place your `license.json` file in a new directory. Using `/home/user/.config/specforge/` is a common practice.
2. Modify the following lines in your `docker-compose-x.y.z.yml` file to point to your license file:

```
- type: bind
  source: path/to/.config/specforge/ # place your license.json file here on the host machine
  target: /app/specforgeconfig/ # config directory inside the container (do not modify this)
  read_only: true
```

> **Note:** It is *not* recommended to run docker as `root` (i.e. with `sudo`). But if you do, note that paths with `~/` would be understood by the system as `/root/`, not your home directory. So it's best to use absolute paths (without `~`).

3. Configure LLM Provider (Optional)

To use LLM-based features such as natural-language spec generation and error explanation, configure an LLM provider by modifying environment variables in your `docker-compose.yml` file before starting the server.

For OpenAI (recommended):

```
- SPECFORGE_LLM_PROVIDER=openai
- SPECFORGE_LLM_MODEL=gpt-5-nano-2025-08-07
- OPENAI_API_KEY=${OPENAI_API_KEY}
```

Get an API key from platform.openai.com/api-keys.

You can insert API keys directly in the file, but using environment variables is better for security.

For other providers (Gemini, Anthropic, Ollama) and detailed configuration options, see the LLM Provider Configuration guide.

4. Start the Server

Run the following command, replacing `/path/to/docker-compose-x.y.z.yml` with the actual path to your downloaded file:

```
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.

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

> **Note**: The server will exit immediately if the license is missing or invalid. If you encounter startup issues, verify your license configuration.

5. Install the VSCode Extension

Install the SpecForge VSCode extension from the Visual Studio Marketplace or see the VSCode Extension setup guide.

Updating the Docker Image

From time to time, new versions of the SpecForge Server are released. To use the latest version, you can either:

1. **Use the updated docker-compose file** from the releases page, or
2. **Set the image field to latest** in your Docker Compose file:

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

   Then pull the latest image:

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

Next Steps

- VSCode Extension - Learn about the VSCode extension features
- Python SDK - Set up the Python SDK for programmatic access
- A Whirlwind Tour - Take a tour of SpecForge capabilities
- Project Configuration - Learn about `lilo.toml` configuration

LLM Provider Configuration

SpecForge includes LLM-based features such as natural-language based spec generation and error explanation. To use these features, you need to configure an LLM provider.

Supported Providers

SpecForge currently supports three LLM providers:

- **OpenAI** - Cloud-based API (recommended for most users)
- **Gemini** - Google's cloud-based API
- **Ollama** - Run models locally on your machine

Configuration Methods

For Executable (Windows, macOS, Linux)

Set the following environment variables before starting the SpecForge server:

OpenAI

```
# Linux / macOS
export SPECFORGE_LLM_PROVIDER=openai
export SPECFORGE_LLM_MODEL=gpt-5-nano-2025-08-07
export OPENAI_API_KEY=your-api-key-here


# Windows PowerShell
$env:SPECFORGE_LLM_PROVIDER="openai"
$env:SPECFORGE_LLM_MODEL="gpt-5-nano-2025-08-07"
$env:OPENAI_API_KEY="your-api-key-here"
```

Get an API key from platform.openai.com/api-keys.

Gemini

```
# Linux / macOS
export SPECFORGE_LLM_PROVIDER=gemini
export SPECFORGE_LLM_MODEL=gemini-2.5-flash
export GEMINI_API_KEY=your-api-key-here


# Windows PowerShell
$env:SPECFORGE_LLM_PROVIDER="gemini"
$env:SPECFORGE_LLM_MODEL="gemini-2.5-flash"
$env:GEMINI_API_KEY="your-api-key-here"
```

Get an API key from ai.google.dev/gemini-api/docs/api-key.

Anthropic

```
# Linux / macOS
export SPECFORGE_LLM_PROVIDER=anthropic
export SPECFORGE_LLM_MODEL=claude-haiku-4-5
export ANTHROPIC_API_KEY=your-api-key-here
```

```
# Windows PowerShell
$env:SPECFORGE_LLM_PROVIDER="anthropic"
$env:SPECFORGE_LLM_MODEL="claude-haiku-4-5"
$env:ANTHROPIC_API_KEY="your-api-key-here"
```

Get an API key from platform.claude.com/docs/en/api/admin/api_keys/retrieve.

Ollama

First, install and run Ollama from docs.ollama.com/quickstart.

Then set the environment variables:

```
# Linux / macOS
export SPECFORGE_LLM_PROVIDER=ollama
export SPECFORGE_LLM_MODEL=your-model-name  # e.g., llama3.2, mistral
export OLLAMA_API_BASE=http://127.0.0.1:11434
```

```
# Windows PowerShell
$env:SPECFORGE_LLM_PROVIDER="ollama"
$env:SPECFORGE_LLM_MODEL="your-model-name"  # e.g., llama3.2, mistral
$env:OLLAMA_API_BASE="http://127.0.0.1:11434"
```

Change `OLLAMA_API_BASE` if your Ollama server is running on a different machine.

For Docker

Modify the environment variables in your `docker-compose.yml` file:

```
- SPECFORGE_LLM_PROVIDER=openai # other options: ollama, gemini
- SPECFORGE_LLM_MODEL=gpt-5-nano-2025-08-07 # choose the appropriate model for your provider
# One of the following, depending on SPECFORGE_LLM_PROVIDER:
- 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
```

You can insert API keys directly in the file:

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

However, it is better to use environment variables for security.

Default Models

If you don't set the `SPECFORGE_LLM_MODEL` variable:

- **OpenAI**: Defaults to `gpt-5-nano-2025-08-07`
- **Gemini**: Defaults to `gemini-2.5-flash`
- **Anthropic**: Defaults to `claude-haiku-4-5`
- **Ollama**: You must specify a model (no default)

Without LLM Configuration

Without an appropriate LLM provider configuration, LLM-based SpecForge features will be unavailable. The rest of SpecForge will continue to work normally.

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

The SpecForge VSCode extension can be installed in two ways:

From the VSCode Marketplace

Install the extension directly from the Visual Studio Marketplace or search for “SpecForge” in VSCode’s extensions tab (Ctrl+Shift+X or Cmd+Shift+X).

From VSIX File

Alternatively, you can install from a VSIX file (included in releases):

- 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

> **Important**: Ensure the extension version matches your SpecForge server version. Version mismatches may cause compatibility issues.

Usage and Configuration

For the extension to work, the SpecForge server must be accessible. There are two ways to connect:

1. **Managed server** (`specforge.spawnServer`): Enable this setting and the extension will start and manage the SpecForge server process for you. Configure `specforge.serverPath` if the `specforge` binary is not on your `PATH` , and `specforge.preferredPort` to choose a port.

2. **External server** (default): Run the SpecForge server yourself (see Setting up SpecForge) and point the extension at it using the `specforge.apiBaseUrl` setting (default: `http://localhost:8080`).

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

The VSCode workspace should be a directory which contains a specific lilo project. The extension will use the `lilo.toml` file at the root of the workspace to determine certain details about the project. Refer to the Python SDK setup guide to see an example of a project structure.

For a full reference of all available settings (LLM integration, server tuning, tracing, etc.), see the Configuration section of the VSCode Extension guide.

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 envionment 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.

The recommended way to use `uv` is to set up a `pyproject.toml` file for *each project separately*. This would allow you to manage dependencies on a per-project basis, and avoid conflicts between different projects. Here is what a typical directory structure would look like:

```
sample_project
├── data
│   ├── first60.csv
│   ├── last60.csv
│   └── sampled.csv
├── lib
│   └── specforge_sdk-0.5.7-py3-none-any.whl
├── scripts
│   └── script.py
├── .venv
│   └── (managed by uv)
├── lilo.toml
├── explore.ipynb
├── main.lilo
├── pyproject.toml
└── uv.lock
```

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:

```toml
[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-0.5.9-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.
    - Note that this `.venv` is unique to this project. Each project should have its own `.venv` directory, meaning that there should be a separate `pyproject.toml` file for each project, and you would have to run `uv sync` for each project separately.
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.

```
$ source .venv/bin/activate
(sample-project) $ which python
/path/to/project/sample-project/.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` configuration file at the project root.

For getting started, you can skip this configuration entirely and simply place your `.lilo` specification files and data files directly in the root of your project. SpecForge will work with sensible defaults.

When you do use `lilo.toml`, 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.

The configuration file is useful for:

- Setting a project name and custom source path (default: project root or `src/`)
- Customizing language behavior (interval mode, freeze)
- Adjusting diagnostics settings (consistency, redundancy, optimize, unused defs) and their timeouts
- Registering `system_falsifier` entries for falsification analysis

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` — SMT solver timeouts for consistency checks:
        - `named` (seconds, float). Timeout per individual named specification. Default: `0.5`
        - `system` (seconds, float). Timeout for the whole-system consistency check (all specs checked together). Default: `1.0`
    - `redundancy.enabled` (bool). Default: `true`
    - `redundancy.timeouts` — SMT solver timeouts for redundancy checks:
        - `named` (seconds, float). Timeout per individual named specification. Default: `0.5`
        - `system` (seconds, float). Timeout for the whole-system redundancy check (all specs checked together). 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/"
```

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"
```

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 φ`: `φ` is true at all future times
- `eventually φ`: `φ` is true at some future time
- `past φ`: `φ` was true at some past time
- `historically φ`: `φ` was true at all past times

These operators can be qualified with time intervals, e.g., `eventually[0, 10] φ` means `φ` 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.

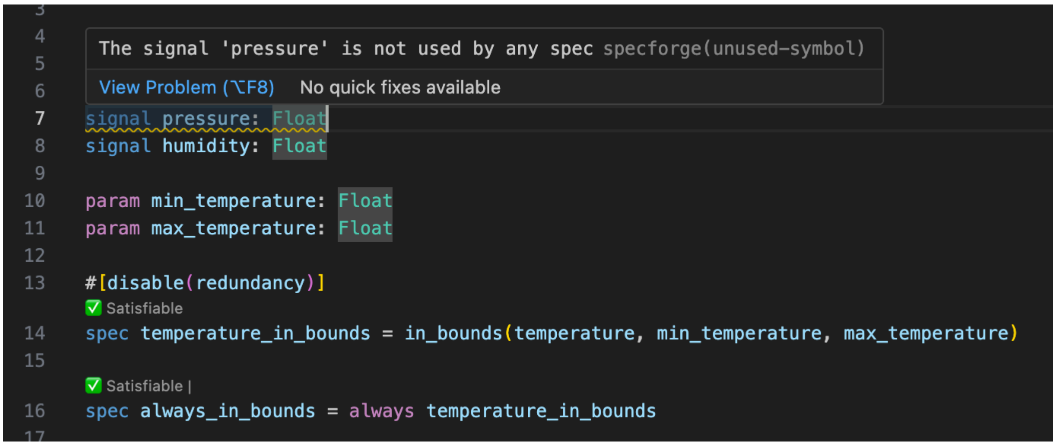
For a comprehensive guide to the language, see the Lilo Language chapter.

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:

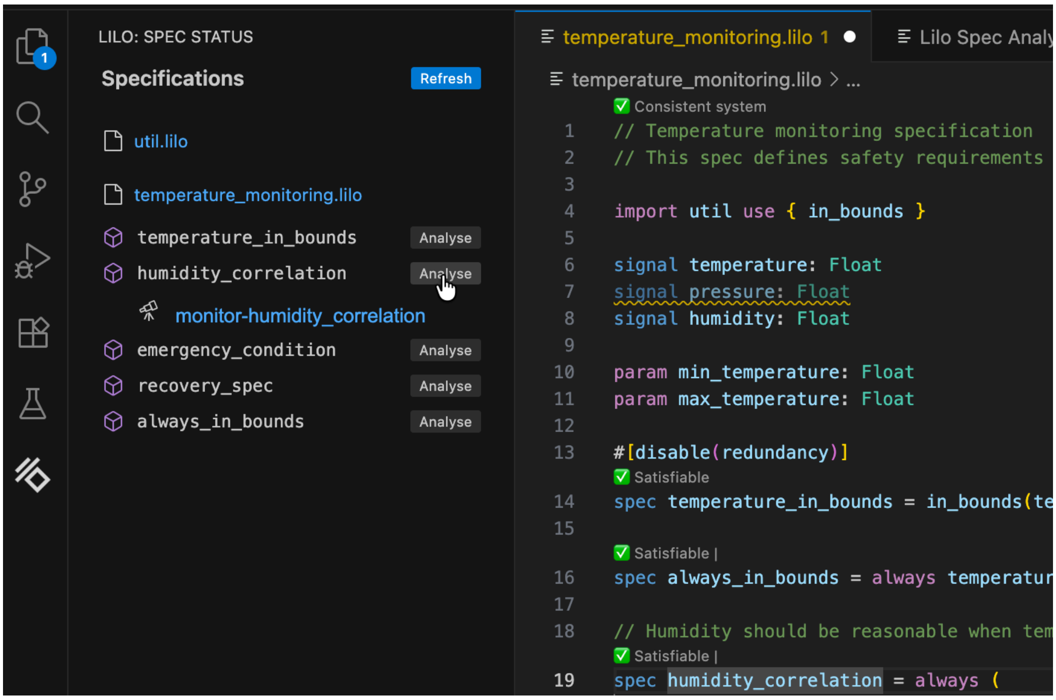
- **Monitor**: Check whether recorded system behavior satisfies specifications
- **Exemplify**: Generate example traces that satisfy specifications
- **Falsify**: Search for counterexamples that violate specifications, relative to some model
- **Export**: Convert specifications to other formats (`.json` , `.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.

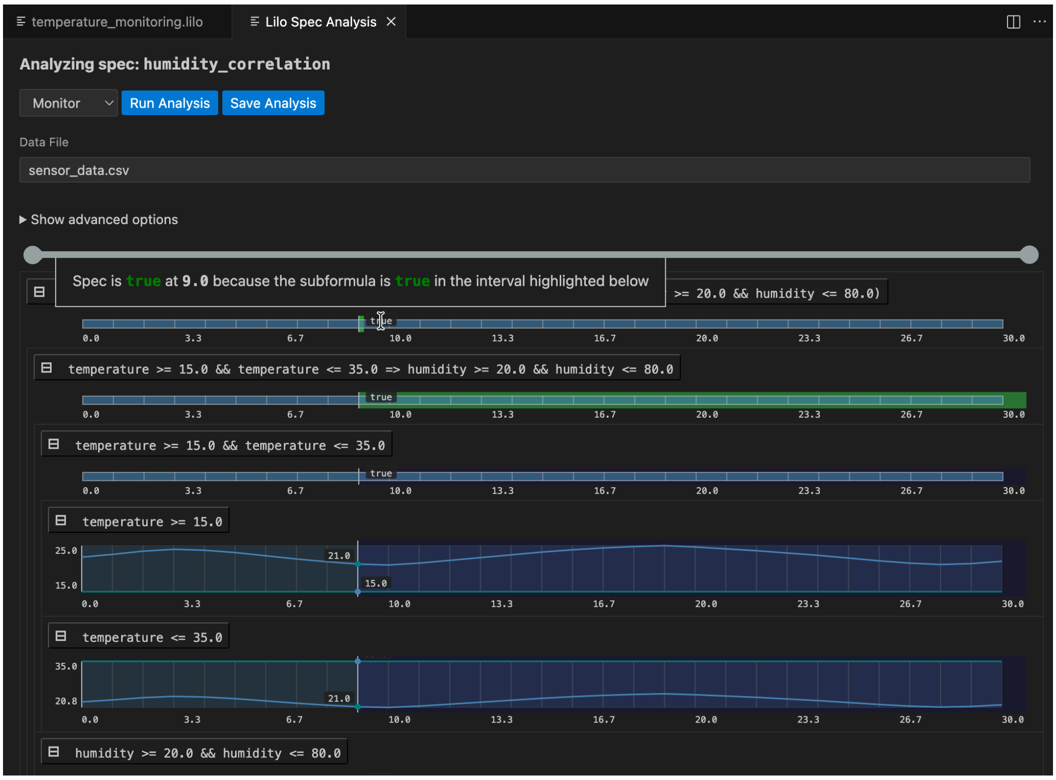
Monitoring

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.

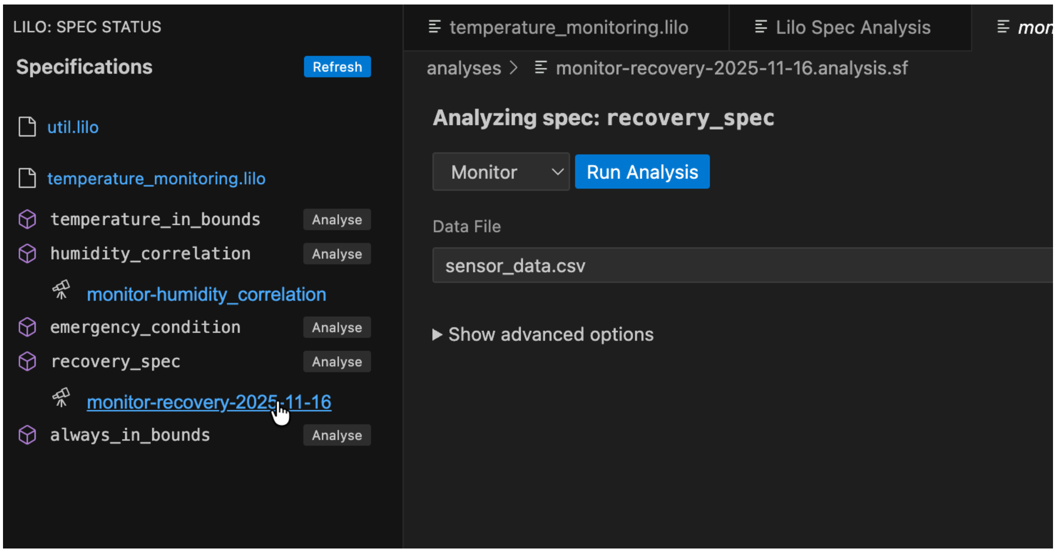


After selecting a data file from the dropdown menu, click `Run Analysis`. The result is an analysis monitoring tree for the specification:



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.

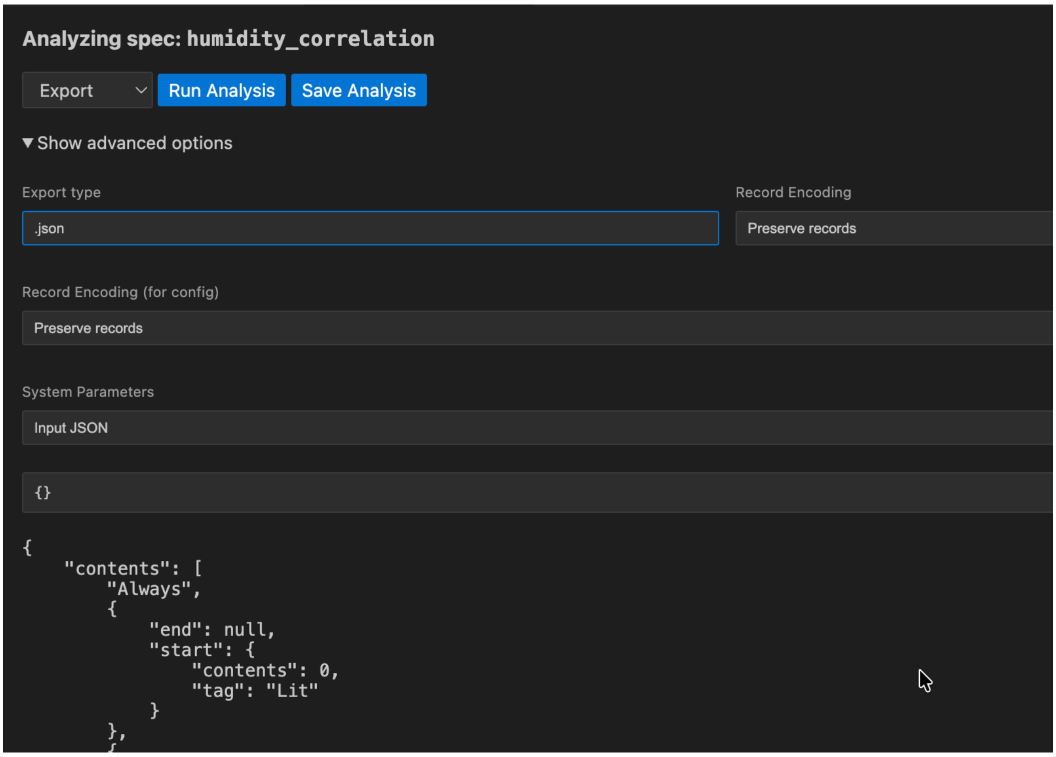
First a falsifier must be registered in `lilo.toml`, e.g.

```
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 show, 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`.



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.

Language Basics

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.

Units of Measure

Lilo supports units of measure for `Float` values. Units are written in angle brackets `<...>` immediately after a literal.

Basic Units

A simple unit is written as an identifier inside angle brackets:

```
1.0<cm>
100.0<km/h>
```

A unit changes the type of the literal from `Float` to a dimensioned type. In the previous example, the type is changed from `Float` to `Float<cm>` or `Float<km/h>` .

Compound Units

Units can be combined using operators. The literal `1` represents a dimensionless unit:

```
60.0<1/s>
```

- **Ratio** (/):

  ```
  15.0<m/s>
  ```

- **Product** (*):

  ```
  50.0<m*m>
  ```

- **Exponentiation** (^):

  ```
  100.0<m^2>
  ```

  ```
  9.81<m*s^-2>
  ```

Operator Precedence and Associativity

Unit operators follow standard mathematical precedence rules:

1. **Exponentiation** (^) has the highest precedence and binds tightly to the immediately preceding unit.
2. **Product** (*) and **ratio** (/) have equal precedence and associate left-to-right.

This means `m/s*kg` is interpreted as `(m/s)*kg`, and `m*s^-2` means `m*(s^-2)`, not `(m*s)^-2`.

Parentheses for Grouping

Parentheses can be used to override the default precedence and associativity:

```
1.0<1/(kg*m)>
```

Operations on Lieterals with Units

You can perform addition, subtraction and comparison on literals with units, such as:

```
100.0<km> + 10.0<km>
50.2<km> - 3.0<km>
6.5<km> > 3.6<km>
```

When performing these operations, the units of the first and second arguments must be the same. Otherwise, a type error will occur.

You can perform multiplication or division even if the units of the first and second arguments are different, such as:

```
100.0<km> * 2.0<s>
100.0<km> / 2.0<s>
```

These are evaluated as `200.0<km*s>` and `50.0<km/s>`, respectively.

Inference

Units can be inferred. In the following example, the type of `speed` is inferred to be `Float<m/s>`.

```
def f(speed) = speed * 5.0<s> <= 10.0<m>
```

Identification

Units are identified as strings. Thus, for example, `m` and `km` have no relation as units. To relate them, for example, define a function to convert from km to m.

```
def km_to_m(kilometre: Float<km>) = kilometre * 1000.0<m/km>
```

Declaring Units

Units used in type annotations must be declared with the `unit` keyword:

```
unit km
unit h
unit s
```

Once declared, units can be used in signal and definition type annotations:

```
signal speed: Float<km/h>
signal distance: Float<km>
```

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 (See Semantics of Temporal Operators for details):
    - `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.
    - `φ releases ψ`: shortcut for `!(!φ until !ψ)`.
    - `next f`: The value of `f` (not necessarily boolean) at the *next* (discrete) time point, or retains its value if there is no next time point.
    - `previous f`: The value of `f` (not necessarily boolean) at the *previous* (discrete) time point, or retains its value if there is no previous time point.
    - `previous_with v f`: The value of `f` (not necessarily boolean) at the previous (discrete) time point, or `v` if there is no previous time point.
    - `next_with v f`: The value of `f` (not necessarily boolean) at the next (discrete) time point, or `v` if there is no next time point.

  One can chain unary temporal operators without parentheses, e.g, `always eventually (x < 0)`.

  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.
    - `φ releases [a, b] ψ`: shortcut for `!(!φ until [a, b] !ψ)`.
    - Similar interval qualifications apply to other temporal operators.
    - One can use `infinity` in intervals: `[0, infinity]`.

- Sliding Window Operators

    - `max_future [0, a] φ`: the maximum value of `φ` in the next `a` time units.
    - `min_future [0, a] φ`: the minimum value of `φ` in the next `a` time units.
    - `max_past [0, a] φ`: the maximum value of `φ` in the past `a` time units.
    - `min_past [0, a] φ`: the minimum value of `φ` in the past `a` time units.
    - Note that the interval must be of the form `[0, a]`.
    - Similar to other temporal operators, one can omit the interval to mean `[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 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.

- `sqrt` returns the square root of a numeric value. The argument must be an `Int` or dimensionless `Float`:

```
sqrt(x)
```

- `abs` returns the absolute value of a numeric value. The argument must be an `Int` or `Float`, and units are preserved:

```
abs(x)
```

- `max` returns the maximum of two or more numeric values. All arguments must have the same type and units:

```
max(x, y)
```

- `min` returns the minimum of two or more numeric values. All arguments must have the same type and units:

```
min(x, y)
```

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.

Case Expressions

If all the branches of a conditional expressions are `Bool`, you can use a `cases` expression.

```
cases {
  temp > 30.0 → eventually temp < 20.0;
  temp < 10.0 → eventually temp > 20.0;
  10.0 ≤ temp ≤ 30.0 → true;
}
```

This is interpreted as `(temp > 30.0) ⇒ (eventually temp < 20.0) && (temp < 10.0 ⇒ eventually temp > 20.0) && (10.0 ≤ temp ≤ 30.0 ⇒ true)`. Thus, the case `10.0 ≤ temp ≤ 30.0 → true` can be omitted. However, we recommend using exhaustive and disjoint conditions for clarity.

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

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)
```

Assumptions

An `assumption` declares a property that is taken as given when analysing the system. Syntactically it is like a `spec` — no parameters, return type is always `Bool` — but it is treated differently by the tooling: assumptions are automatically included as constraints during exemplification and satisfiability checking, rather than being something to verify.

```
signal temperature: Float
signal heater_on: Bool

assumption physics = always (heater_on ⟹ next temperature ⩾ temperature)

spec eventually_warm = eventually (temperature > 30.0)
```

In this example, `physics` must hold for any example traces SpecForge generates, or when checking satisfiability for specs. See Exemplification and Satisfiability for more details on how assumptions interact with analysis.

Modules

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)
```

The name of the module must match the file name. For example, a module declared as `module Util` must be defined in a file named `Util.lilo`.

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
```

To import units from another module, prefix each unit with the `unit` keyword in the `use` list:

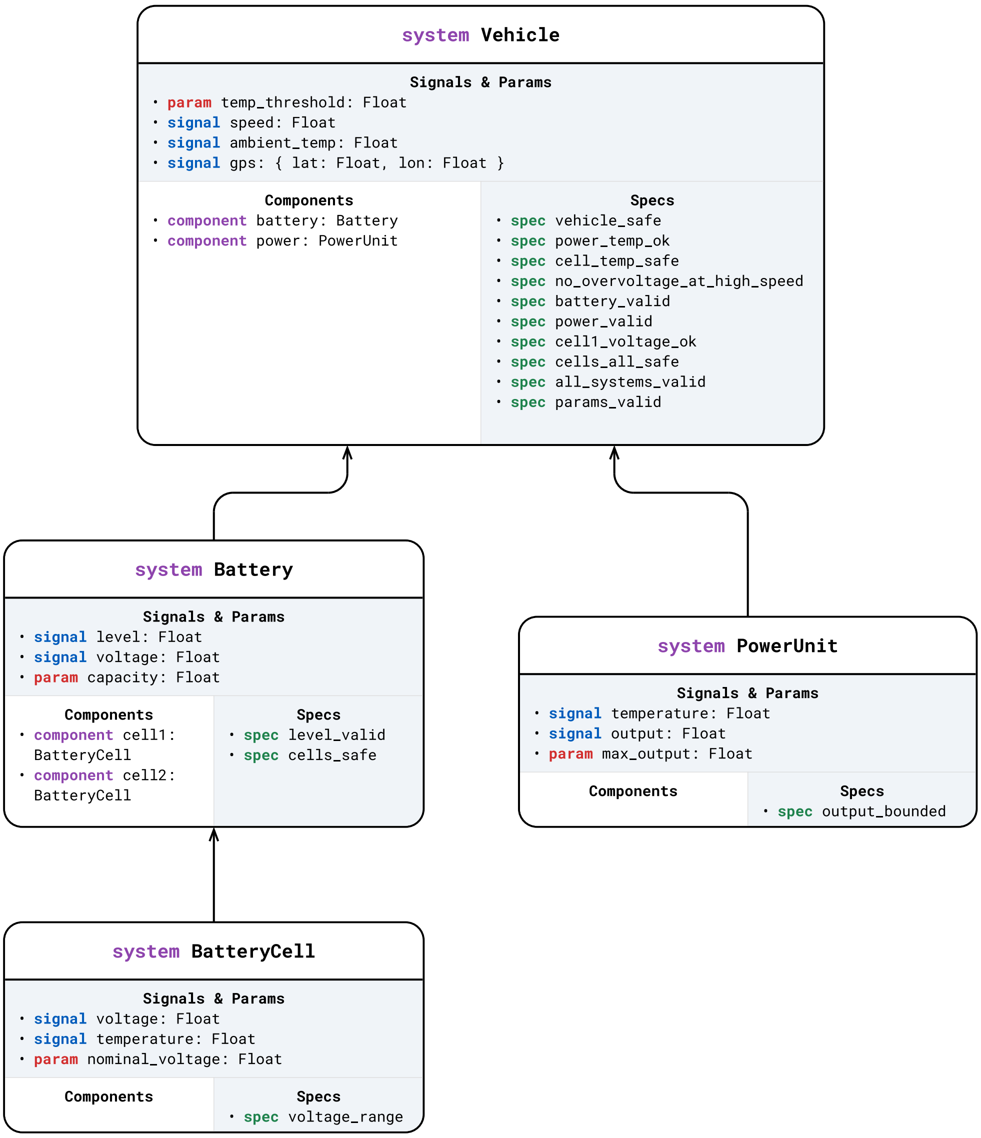
```
import Units use { unit m, unit s }
```

Components

Components are a way of organizing Lilo Systems in hierarchical building blocks.

Consider the following `Vehicle` system, which is organized in the following way:

- A `Vehicle` has a `PowerUnit` and a `Battery`.
- The battery consists of two `BatteryCell` s.
- Each of the systems `PowerUnit`, `Battery`, `BatteryCell` and `Vehicle` have their own signals, parameters and specifications.



The systems `BatteryCell` and `PowerUnit` do not have any components, so they can be declared in a straightforward way:

```
// BatteryCell.lilo
system BatteryCell

pub signal voltage: Float
pub signal temperature: Float
pub param nominal_voltage: Float

pub def over_voltage: Bool = voltage > nominal_voltage * 1.15

spec voltage_range = voltage ⩾ 2.5 && voltage ⩽ 4.5

// PowerUnit.lilo
system PowerUnit

pub signal temperature: Float
pub signal output: Float
pub param max_output: Float

pub def is_overheating: Bool = temperature > 85.0

spec output_bounded = output ⩽ max_output
```

Here, the use of the `pub` keyword indicates that these `signal` s, `param` s and `def` initions will be accessible to other systems which instantiate these systems as components. Specifications declared with `spec` are always public.

Instantiating Components

The `Battery` system uses `BatteryCell` components, which can be declared using the `component` keyword:

```
// Battery.lilo

system Battery

pub signal level: Float
pub signal voltage: Float
pub param capacity: Float

component cell1: BatteryCell

// ...
```

Lifted vs Mapped Signals and Params

When a system is instantiated as a component in another system, its params and signals are **lifted** to the parent system. Thus, in addition to `level` , `voltage` and `capacity` , the schema for `Battery` involves `cell1::voltage` , `cell1::temperature` and `cell1::nominal_voltage` .

Some signals or params of a component may be **mapped**, meaning that they are not a part of the input schema, but their values are determined by other signals or params (of the parent system). For example, consider the following example where the schema of `Battery` still exposes `cell1::voltage` , `cell1::temperature` and `cell1::nominal_voltage` , as well as `cell2::temperature` , but not `cell2::nominal_voltage` or `cell2::voltage` . The value of `cell2::nominal_voltage` is fixed to `3.7` , and the value of `cell2::voltage` is derived from the `Battery` 's `voltage` signal.

```
// Unmapped component - all signals and params will be lifted
component cell1: BatteryCell

// Partially mapped component - voltage is mapped, temperature is lifted
component cell2: BatteryCell {
  param nominal_voltage = 3.7
  signal voltage = voltage / 2.0  // Derived from battery voltage
}
```

Accessing Component Signals and Params

Lilo expressions in a system can refer to signals and params of their components using the `::` operator to access the component's signals and params.

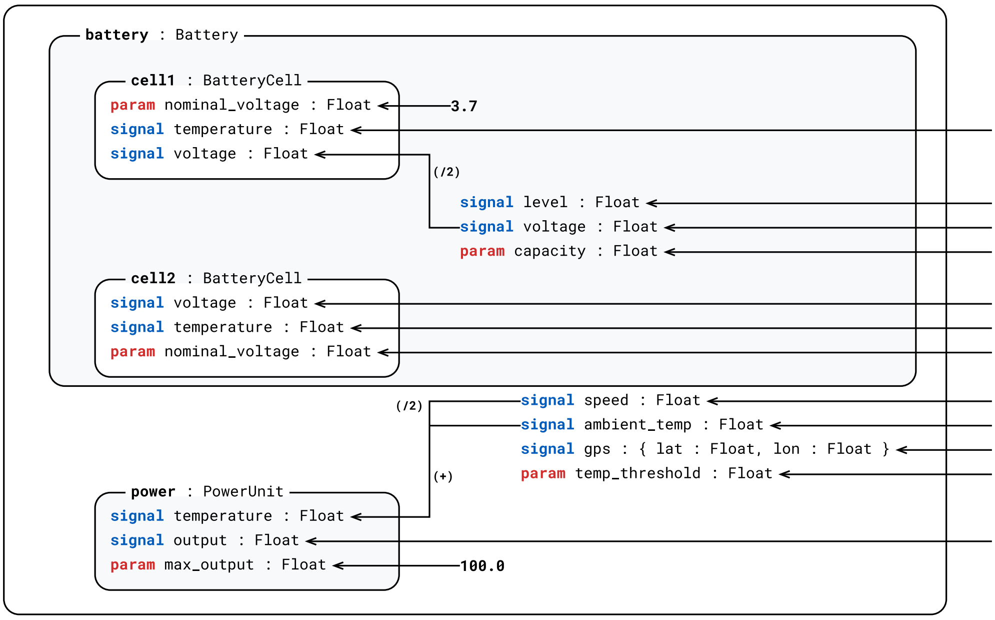
```
// Use cell component signals
pub def any_cell_over_voltage: Bool = cell1::over_voltage || cell2::over_voltage

spec level_valid = level ⩾ 0.0 && level ⩽ 100.0

// Reference cell component specs
spec cells_safe = cell1::voltage_range && cell2::voltage_range
```

A system can refer to signals and params of components of its components. For example, consider the `Vehicle` system which has both a `Battery` and a `PowerUnit` component. One can write specifications in the `Vehicle` system in the following manner.

```
// Vehicle.lilo

// ...

// Two-level signal references

def low_battery: Bool = battery::level < 20.0
def power_hot: Bool = power::temperature > 80.0

def bat_cell1_voltage: Float = battery::cell1::voltage

def at_meihan: Bool = 34.63 < gps.lat < 34.65 && 135.99 < gps.lon < 136.01

// Three-level signal references (Vehicle → Battery → BatteryCell)

def cell1_overheated: Bool = battery::cell1::temperature > 65.0
def cell2_voltage_ok: Bool = battery::cell2::voltage > 3.0

// Two-level def references

def battery_or_power_issue: Bool = battery::is_low || power::is_overheating

// Three-level def references (Vehicle → Battery → BatteryCell)

def any_cell_overvoltage: Bool = battery::cell1::over_voltage || battery::cell2::over_voltage

// Combined two-level and three-level
def critical_state: Bool = battery::is_low && battery::any_cell_over_voltage

// ...
```

System Schema

The **schema** of a system refers to the set of signals and params that constitutes it input. When monitoring, the user provides values for elements of the schema. The schema corresponding to the `Vehicle` system is shown in the diagram below.



The SpecForge CLI `schema` command can be used to view the schema of a system. Notice that the mapped signals and params do not appear in the schema, since their values are derived from different signals or params.

```
$ schema --project /Users/agnishom/code/reqeng/api/examples/projects/vehicle -s Vehicle flat
Signals:
  ambient_temp  Float
  battery::cell1::temperature  Float
  battery::cell1::voltage  Float
  battery::cell2::temperature  Float
  battery::level  Float
  battery::voltage  Float
  gps_lat  Float
  gps_lon  Float
  power::output  Float
  speed  Float


Params:
  battery::capacity  Float
  battery::cell1::nominal_voltage  Float
  temp_threshold  Float   (default: 75.0)
```

Refer to the documentation on Data Files for details on the format for providing values for the schema when monitoring.

Static Analysis

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 `positive_becomes_negative` we conclude that therefore there must be some point in time after than when `x < 0`.

Guard Analysis for Case Expressions

If a spec is a case expression, we check the guards in the case expression for satisfiability, exhaustiveness, and disjointness.

```
spec example = cases {
    guard1 -> consequence1;
    guard2 -> consequence2;
    ...
}
```

- Satisfiability: Whether each guard is satisfiable on its own.
- Exhaustiveness: Whether the guards taken together cover all possible cases.
- Disjointness: Whether the guards are mutually exclusive.

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.
- Soft assumptions: `#[rigidity = "soft"]`
  - Applied to an `assumption`, marks it as a soft constraint that the solver tries to satisfy but may relax if it conflicts with hard constraints. See Rigidity Levels for details.

Labels, Aliases, and Custom Fields

Specifications can be annotated with labels, aliases, and custom fields, using attributes.

A label is a string attached to a specification. The VSCode extension's spec sidebar groups specifications by label. This is useful for focusing on a subset of specifications, such as the safety-related ones.

```
#[label("safety", "critical")]
spec brake_must_work = always (brake_pedal => eventually (deceleration > 0))
```

An alias is an alternative name in a particular language. The spec sidebar shows the alias alongside the name used in the source file.

```
#[alias(en = "Brake Must Work", ja = "ブレーキは動作しなければならない")]
spec brake_must_work = always (brake_pedal => eventually (deceleration > 0))
```

A specification can have multiple `#[label]` or `#[alias]` attributes. All of them are applied.

A label can be associated with a color (represented via a hex code or any standard HTML5 color name) in the `lilo.toml` config file. To do so, simply add a section `[labels.colors]` like the following:

```
[labels.colors]
production = "blue"
consumption = "magenta"
renewable = "0x00ff00"
'mitigation strategy' = "yellow"
```

Custom fields attach extra scalar metadata to a specification. They are useful for project-specific categorization such as review status, owner, priority, etc.

```
#[field(priority = 1, reviewed = true, owner = "ops")]
spec brake_must_work = always (brake_pedal => eventually (deceleration > 0))
```

Custom field values must be scalar values (strings, numbers, booleans). For a single specification, if the same custom field is provided more than once, the first value is used and a warning is emitted.

The VSCode extension can use custom fields in the Spec Status Pane filter. For example:

- `reviewed:true`
- `owner:"ops"`
- `priority:>=2`

See VSCode Extension for the full query syntax.

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.

```
#[default = 25.0]
param temperature: Float
```

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
#[default = 1]
param bound: Int

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
```

Stubs can also be used for assumptions which have not yet been formalized.

```
/// height is always non-negative
assumption height_non_negative
```

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`.
- **Important**: The name of a module or system must match the file name it is defined in. For example, `module climate_control` or `system climate_control` must be defined in a file named `climate_control.lilo`.
- 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
}
```

Semantics of Temporal Operators

Lilo adopts a semantics for temporal logic that is slightly non-standard, in the sense that it takes into account both a discrete and a continuous notion of time.

Let `T` be a type. A sample of type `T` is a pair `{time : Real, value : T}`. A signal of type `T` is a finite sequence of at-least two samples of type `T` such that the time values are strictly increasing. We write `Signal[T]` to be the set of all signals of type `T`. We do not require the first sample of a signal to be at time `0`. Given a signal `σ`, we denote the elements of `σ` as `σ[0]`, `σ[1]`, ..., `σ[n-1]` where `n` is the number of samples of `σ`. Note that we distinguish between the number of samples of `σ` and the duration of `σ` (which is `σ[n-1].time - σ[0].time`).

The support of a signal `σ`, denoted `support(σ)`, is the set of time values associated with the samples of `σ`, i.e., `support(σ) = {σ[i].time | 0 ≤ i < n}`. We write `σ[time = t]` to denote the value of `σ` at time `t`, i.e, `σ[time = t] = σ[i].value` where `i` is such that `σ[i].time = t`. Note that `σ[time = t]` is only defined for `t ∈ support(σ)`.

Each Lilo formula `φ` is judged to be of some type `T` (written `φ : T`), by the Lilo type system. The semantics of a Lilo formula of type `T` is understood as a function of type `Signal[S] → Signal[T]`, where `S` is the type of the input signals described by the system. This function is defined to be support-preserving, i.e, the output signal is defined on exactly the same time values as the input signal. For a formula `φ` of type `T`, we write `⟦φ⟧` to denote this function.

Atemporal operators include the standard Boolean operators, arithmetic operators, comparison operators, conditionals and so on. A formula is said to a *pointwise formula* if for any fixed time value `t`, the value of the output signal `⟦φ⟧ (σ)[time = t]` depends only on the value of the input signal `σ[time = t]`. A lilo expression is said to be atemporal if its intepretation does not depend on the input signals. Thus, a pointwise formula is built from input signals and atemporal operators, and an atemporal expression is built from params, constants and atemporal operators.

An interval of real numbers is written as a pair `[a, b]` where `a: Real` and `b: Real | infinity` with `a ≤ b`. If `b` is `infinity`, we say that the interval is unbounded, and otherwise we say that the interval is bounded. For a time point `t` and an interval `I`, we write `t + I` or `t - I` to denote `{t + x | x ∈ I}` or `{t - x | x ∈ I}` respectively. In our case, since each of the temporal operators preserves support, we will write `t + I` and `t - I` as a shorthand for `(t + I) ∩ support(σ)` and `(t - I) ∩ support(σ)`.

Temporal operators in Lilo, with some exceptions, are annotated with an interval. Given `φ : Bool`, we define the following temporal operators.

- **Always:** `⟦always[I] φ⟧ (σ)[time = t] = true` if and only if for all `t' ∈ (t + I)`, `⟦φ⟧ (σ)[time = t'] = true`.
- **Eventually:** `⟦eventually[I] φ⟧ (σ)[time = t] = true` if and only if there exists `t' ∈ (t + I)` such that `⟦φ⟧ (σ)[time = t'] = true`.
- **Historically:** `⟦historically[I] φ⟧ (σ)[time = t] = true` if and only if for all `t' ∈ (t - I)`, `⟦φ⟧ (σ)[time = t'] = true`.
- **Past:** `⟦past[I] φ⟧ (σ)[time = t] = true` if and only if there exists `t' ∈ (t - I)` such that `⟦φ⟧ (σ)[time = t'] = true`.
- **Until:** `⟦φ until[I] ψ⟧ (σ)[time = t] = true` if and only if there exists `t' ∈ (t + I)` such that `⟦ψ⟧ (σ)[time = t'] = true` and for all `t'' ∈ support(σ)` satisfying `t ≤ t'' < t'`, `⟦φ⟧ (σ)[time = t''] = true`.
- **Since:** `⟦φ since[I] ψ⟧ (σ)[time = t] = true` if and only if there exists `t' ∈ (t - I)` such that `⟦ψ⟧ (σ)[time = t'] = true` and for all `t'' ∈ support(σ)` satisfying `t' < t'' ≤ t`, `⟦φ⟧ (σ)[time = t''] = true`.

Note that the `always` and `eventually` operators are logical duals of each other. The logical dual of `until` is `releases`, i.e, `φ releases[I] ψ` is defined as `!(!φ until[I] !ψ)`. The operators `since`, `historically` and `past` are the past-time counterparts of `until`, `always` and `eventually` respectively.

The operators `will_change` and `did_change` are defined for any well-typed `φ : T` for which equality is defined (which is currently all non-function types) as follows.

- **Will Change:** `⟦will_change[I] φ⟧ (σ)[time = t] = true` if and only if there exists two points `t', t'' ∈ (t + I)` such that `⟦φ⟧ (σ)[time = t'] ≠ ⟦φ⟧ (σ)[time = t'']`.
- **Did Change:** `⟦did_change[I] φ⟧ (σ)[time = t] = true` if and only if there exists two points `t', t'' ∈ (t - I)` such that `⟦φ⟧ (σ)[time = t'] ≠ ⟦φ⟧ (σ)[time = t'']`.

The following operators cannot be annotated with an interval. And importantly, they are defined on the discrete time-semantics, i.e, they depend on the discrete index associated with the samples rather than the associated `time`. If the base formula `φ` is of type `T`, and `v` is an atemporal value of type `T`, then so are the formulas `next φ`, `previous φ`, `next_with v φ` and `previous_with v φ`. We write `n` to denote the number of samples of the input signal `σ`.

- **Next:** `⟦next φ⟧ (σ)[i] = ⟦φ⟧ (σ)[i + 1]` for `0 ≤ i < n - 1` and `⟦next φ⟧ (σ)[n - 1] = ⟦φ⟧ (σ)[n - 1]`.
- **Previous:** `⟦previous φ⟧ (σ)[i] = ⟦φ⟧ (σ)[i - 1]` for `0 < i < n` and `⟦previous φ⟧ (σ)[0] = ⟦φ⟧ (σ)[0]`.
- **Next With**: `⟦next_with v φ⟧ (σ)[i] = ⟦φ⟧ (σ)[i + 1]` for `0 ≤ i < n - 1` and `⟦next_with v φ⟧ (σ)[n - 1] = v`.
- **Previous With**: `⟦previous_with v φ⟧ (σ)[i] = ⟦φ⟧ (σ)[i - 1]` for `0 < i < n` and `⟦previous_with v φ⟧ (σ)[0] = v`.

The following *sliding window* operators are defined for any `φ` whose value is a numeric type. Note also that the associated interval must be of the form `[0, b]` where `b` is possibly `infinity`.

- **Future Maximum:** `⟦max_future [0, a] φ⟧ (σ)[time = t] = max{⟦φ⟧ (σ)[time = t'] | t' ∈ support(σ), t ≤ t' ≤ t + a}`.
- **Past Maximum:** `⟦max_past [0, a] φ⟧ (σ)[time = t] = max{⟦φ⟧ (σ)[time = t'] | t' ∈ support(σ), t - a ≤ t' ≤ t}`.
- **Future Minimum:** `⟦min_future [0, a] φ⟧ (σ)[time = t] = min{⟦φ⟧ (σ)[time = t'] | t' ∈ support(σ), t ≤ t' ≤ t + a}`.
- **Past Minimum:** `⟦min_past [0, a] φ⟧ (σ)[time = t] = min{⟦φ⟧ (σ)[time = t'] | t' ∈ support(σ), t - a ≤ t' ≤ t}`.

Exemplification and Satisfiability

Exemplification is the process of generating example traces that satisfy a specification.

Satisfiability is a closely related concept: a spec is satisfiable if there exists at least one trace that makes it true. SpecForge's static analysis automatically checks whether your specs are satisfiable and warns you if they are not.

Both exemplification and satisfiability checking are influenced by **assumptions**, which are additional constraints that the solver takes into account.

Assumptions

An assumption is a property that is taken for granted when analysing a specification. Instead of being something you want to verify, it represents background knowledge about the system: physical laws, environmental constraints, or operating conditions that are expected to hold.

The `assumption` Keyword

In Lilo, you can declare an assumption using the `assumption` keyword:

```
signal temperature: Float
signal heater_on: Bool

assumption physics = always (heater_on ⟹ next temperature ⩾ temperature)

spec eventually_warm = eventually (temperature > 30.0)
```

An `assumption` is syntactically like a `spec` (it has no parameters and its return type is always `Bool`) but it is treated differently by the tooling:

- **Exemplification**: When generating example traces for any spec in the same system, all assumptions are automatically included as constraints. The solver must produce traces that satisfy both the spec and every assumption.
- **Satisfiability**: When checking whether a spec is satisfiable, assumptions are included in the constraint set. A spec that would be satisfiable on its own may become unsatisfiable under assumptions.
- **Redundancy**: Assumptions are included alongside other specs when checking for redundancy. A spec that follows from the assumptions (and other specs) will be flagged as redundant.

You can add the `#[rigidity = "soft"]` attribute to an assumption to make it a soft constraint for the exemplifier. See the section on Rigidity Levels below for details. By default, assumptions are considered hard.

This means you do not need to manually pass assumptions when running analyses. Declaring them in your Lilo source is enough.

Ad-hoc Assumptions

In addition to the `assumption` keyword, you can pass **ad-hoc assumptions** when running exemplification through the Python SDK or the VSCode extension. These are one-off constraints that apply only to a specific analysis run, rather than being part of the system definition.

Ad-hoc assumptions are useful when you want to explore "what if" scenarios without modifying your Lilo source, for example "generate a trace where the temperature starts above 20".

Rigidity Levels

Each assumption (whether declared with the `assumption` keyword in the source or passed ad-hoc to an exemplification query) has a **rigidity** level that controls how the solver treats it:

- **Hard**: The assumption is treated as an inviolable constraint. The generated trace **must** satisfy the assumption. If the solver cannot find a trace that satisfies both the spec and all hard assumptions, exemplification fails.
- **Soft**: The assumption is treated as a preference. The solver first attempts to find a trace satisfying the spec and all hard assumptions, then tries to additionally satisfy soft assumptions. If a soft assumption cannot be satisfied together with the hard constraints, it is relaxed and the solver still produces a result.

In practice, use **Hard** for constraints that are essential (e.g., physical laws, safety bounds) and **Soft** for constraints that are desirable but not strictly required (e.g., “the temperature should increase at some point”).

> **Note:** Assumptions declared with the `assumption` keyword default to hard rigidity. To make a keyword-declared assumption soft, annotate it with `#[rigidity = "soft"]`.

Fixing Params to Default Values

When running an exemplification task, Lilo will automatically fix any parameters to their default values, if the Lilo source file supplies them. This means that the exemplifier will be forced to find a model in which the parameters take on those specific values.

In some cases, this can lead to unsatisfiability if the default values are incompatible with the spec or assumptions. In the *easy analysis* mode on the VSCode extension, you can disable this behavior by unchecking the “Fix Params to Defaults” option. In this case, the exemplifier will behave as if the parameters had no default values. In the Python SDK, you can achieve the same effect by passing `fix_defaults = False` to the `exemplify` method.

If you want to fix most params to their default values but allow some specific ones to vary, this can be done by passing the `null` JSON value for those params when running exemplification.

To consider a concrete example, consider the following system:

```
system VendingMachine

#[default = 1]
param cansMax : Int

signal cans : Int

assumption cansPositive = cans ≥ 0

spec reasonableCans = 0 < cans < cansMax
```

This spec is not satisfiable when `cansMax` is 1, which is its default value, since there is no integer `cans` satisfying `0 < cans < 1`. Thus, this will be found unsatisfiable if the “Fix Params to Defaults” option is checked. If the option is unchecked, however, the exemplifier is free to choose a different value for `cansMax` (e.g., 5) so that the spec becomes satisfiable. Another way to achieve the same effect is to pass `{ cansMax: null }`, which signals to the exemplifier that `cansMax` should be treated as a free variable.

Example

Consider a temperature sensor system with a spec `always_in_bounds`. You might run exemplification with a mix of rigidity levels:

- A **hard** assumption `"always_in_bounds"` ensures the generated trace respects the bounds. The solver will fail if this is impossible.
- A **soft** assumption `"eventually (temperature > 30)"` expresses a preference that the trace includes a high temperature reading, but the solver will still produce a result if this conflicts with the hard constraints.

Data Files

SpecForge supports three data formats for signal data: CSV, JSON, and JSONL (also known as NDJSON). Parameters are provided separately in a JSON file.

Signal data

Each data sample has a `time` field and one or more signal values. The `time` field is required and must be named exactly `time` (case-sensitive). It accepts any numeric value (integer or floating point) and samples should be in increasing time order.

CSV

Each column header maps to a signal name. Record-typed signals are flattened with a separator (default `_`).

```
time,speed,ambient_temp,gps_lat,gps_lon
0.0,0.0,25.0,34.634,135.996
1.0,30.0,26.0,34.635,135.997
```

Booleans in CSV can be written as `true` / `false`, `True` / `False`, or `1` / `0`.

JSON

A JSON array of objects, each with `time` and `value` keys. Record-typed signals can use nested JSON objects:

```json
[
  {
    "time": 0.0,
    "value": {
      "speed": 0.0,
      "ambient_temp": 25.0,
      "gps": { "lat": 34.634, "lon": 135.996 }
    }
  },
  {
    "time": 1.0,
    "value": {
      "speed": 30.0,
      "ambient_temp": 26.0,
      "gps": { "lat": 34.635, "lon": 135.997 }
    }
  }
]
```

They can also be flattened (see Record encoding):

```json
[
  {
    "time": 0.0,
    "value": {
      "speed": 0.0,
      "ambient_temp": 25.0,
      "gps_lat": 34.634,
      "gps_lon": 135.996
    }
  },
  {
    "time": 1.0,
    "value": {
      "speed": 30.0,
      "ambient_temp": 26.0,
      "gps_lat": 34.635,
      "gps_lon": 135.997
    }
  }
]
```

JSONL

Same as JSON, but one object per line without the outer array.

```
{"time": 0.0, "value": {"speed": 0.0, "ambient_temp": 25.0, "gps": {"lat": 34.634, "lon": 135.996}}}
{"time": 1.0, "value": {"speed": 30.0, "ambient_temp": 26.0, "gps": {"lat": 34.635, "lon": 135.997}}}
```

Column names

The column names in a data file must match the signal declarations in the spec. How signal names map to column names depends on two things: the component hierarchy and the record encoding.

> **Note**: Extra columns in data files that do not match any signal declaration are silently ignored.

Component hierarchy

Signals from system components are prefixed with the component path, separated by `::`. For instance, given a `Battery` system:

```
system Battery

signal level: Float
signal voltage: Float
```

And a `Vehicle` that uses it:

```
system Vehicle

signal speed: Float

component battery: Battery
```

The data file needs columns for `speed` (a direct signal) and `battery::level`, `battery::voltage` (signals from the `Battery` component).

In CSV this looks like:

```
time,speed,battery::level,battery::voltage
0.0,0.0,80.0,7.4
```

In JSON, components can also be written as nested objects:

```
{
  "time": 0.0,
  "value": {
    "speed": 0.0,
    "battery": { "level": 80.0, "voltage": 7.4 }
  }
}
```

Record encoding

Signals with record types need their fields represented as separate columns. There are two encoding modes.

Flat (default): record fields are concatenated with a separator (default `_`). This applies to all formats. For example, given:

```
signal gps: { lat: Float, lon: Float }
```

The columns are `gps_lat` and `gps_lon`.

```
time,gps_lat,gps_lon
0.0,34.634,135.996
```

A custom separator can be specified. Forbidden characters differ by format:

- **CSV**: the separator cannot contain `,` (the CSV delimiter) or `::` (the qualified-name separator).
- **JSON/JSONL**: the separator cannot contain `"` (string-key delimiter), `\` (escape character), or `::` (the qualified-name separator).

Nested: record fields are represented as nested JSON objects. This only applies to JSON and JSONL.

```
{
  "time": 0.0,
  "value": { "gps": { "lat": 34.634, "lon": 135.996 } }
}
```

Mapped signals

When instantiating a component, any of its signals (or parameters) can be *mapped* to an expression. A mapped signal is computed from other signals in the parent system rather than read from data, so it does **not** need a column in the data file.

For example, given a `PowerUnit` with two signals:

```
system PowerUnit

param max_output: Float

signal output: Float
signal temperature: Float
```

A `Vehicle` can map `temperature` to an expression:

```
system Vehicle

signal speed: Float
signal ambient_temp: Float

component power: PowerUnit {
  param max_output = 100.0
  signal temperature = ambient_temp + speed * 0.5
}
```

Here `power::temperature` is computed from the parent signals `ambient_temp` and `speed`, so it does **not** need a column in the data file. Similarly, `power::max_output` is mapped to `100.0` and does not need to appear in the parameter file. Only unmapped signals like `power::output` require data columns.

Parameter files

The structure of a parameter file mirrors the param declarations in the spec.

The `Vehicle` system declares its own parameter and provides a value for `PowerUnit`'s:

```
system Vehicle

param temp_threshold: Float

component power: PowerUnit {
  param max_output = 100.0
}
```

A corresponding parameter file:

```
{
  "temp_threshold": 50.0
}
```

Parameters with default values can be omitted. In the example above, `power::max_output` has a default of `100.0` and is not present in the file.

Putting it all together

Assembling the `Battery` and `PowerUnit` systems from above:

```
system Vehicle

signal speed: Float
signal ambient_temp: Float
signal `fuel level`: Float
signal gps: { lat: Float, lon: Float }

param temp_threshold: Float

component battery: Battery
component power: PowerUnit {
  param max_output = 100.0
  signal temperature = ambient_temp + speed * 0.5
}
```

The parameter file provides `temp_threshold` and omits `power::max_output` (it has a default):

```
{
  "temp_threshold": 50.0
}
```

A CSV data file for this system (flat encoding, `_` separator):

```
time,speed,ambient_temp,fuel
level,gps_lat,gps_lon,battery::level,battery::voltage,power::output
0.0,0.0,25.0,100.0,34.634,135.996,80.0,7.4,0.0
1.0,30.0,26.0,99.8,34.635,135.997,78.0,7.3,40.0
```

Note that `` `fuel level` `` appears as `fuel level` without backticks, and `power::temperature` is absent because it is a mapped signal.

The same data in JSON (nested encoding):

```json
[
  {
    "time": 0.0,
    "value": {
      "speed": 0.0,
      "ambient_temp": 25.0,
      "fuel level": 100.0,
      "gps": { "lat": 34.634, "lon": 135.996 },
      "battery": { "level": 80.0, "voltage": 7.4 },
      "power": { "output": 0.0 }
    }
  },
  {
    "time": 1.0,
    "value": {
      "speed": 30.0,
      "ambient_temp": 26.0,
      "fuel level": 99.8,
      "gps": { "lat": 34.635, "lon": 135.997 },
      "battery": { "level": 78.0, "voltage": 7.3 },
      "power": { "output": 40.0 }
    }
  }
]
```

Notes

- Integer values are automatically coerced to Float when the signal type is `Float`.
- The data format is auto-detected from the file extension (`.csv`, `.json`, `.jsonl`) but can be overridden with the `--file-format` CLI flag.

Falsification

Falsification is the discovery of inputs to a system which violates the given specifications. Generally, the falsification problem consists of the following.

- Context: System Model (i.e, a relation between input signals and output signals)
- Input: A Specification on the input and output signals of the system
- Output: An Input Signal that causes the system to violate the specification

Falsification tools can be categorized as follows.

- **System-Dependent:** A falsifier which is designed to work with a fixed system model.
- **System-Independent:** A falsifier which can be used with any system model, i.e, the falsifier allows the user to choose a model.
  - **Black-box:** A system-independent falsifier where the model can only be understood by the falsifier through its inputs and outputs.
  - **Glass-box:** A system-independent falsifier where the falsifier accepts a description of the model rather than a handle to execute it.

SpecForge currently only supports interfacing with *System-Dependent* falsifiers. This means that the user must supply a script which fixes both the model and provides the falsifier (which is specific to the model).

Below, we describe the details of how to set up such a falsifier to interface with SpecForge. We refer the user to the falsification example projects for complete examples, which we include in the SpecForge releases page.

- **F16 Aircraft Model**: This is a model of the F16 aircraft modelled using 16 continuous variables with piecewise nonlinear differential equations, whose core component is a ground collision avoidance system.
- **Automatic Transmission** (requires MATLAB): This is a model that selects a gear in a vehicle, which has two inputs (a throttle and a brake). Other components of the model include the speed and rotations per minute. This model is inspired by the one commonly included as a part of MATLAB documentation examples, and is implemented using Simulink.

Preparing the Falsification Script

The falsification script must be able to accept the following arguments.

- The path to the SpecForge project directory (i.e, where the `lilo.toml` file is located) via `--project-dir`
- The name of the system to falsify via `--system`
- The name of the specification to falsify via `--spec`
- Values for `param` s of the system in the form of a JSON via `--params`
- Additional options for the script via `--options`

Thus, the command would be invoked as follows.

```
sh scripts/automatic_transmission.sh --system 'automatic_transmission' --spec 'AT6a' --
options '{}' --params '{}' --project-dir './spec/'
```

The output of the falsification script should either be

- `{ "tag": "Err", "contents": err}`
- `{ "tag": "Ok", "contents": { "signal": signal}}`

Where `err` is a string (such as "Couldn't Falsify"), and `signal` is the result of the falsification in JSON format. If using a pandas dataframe, this could be obtained using the `converters.python_to_api_timeseries` function in the SpecForge SDK.

Registering the Falsifier

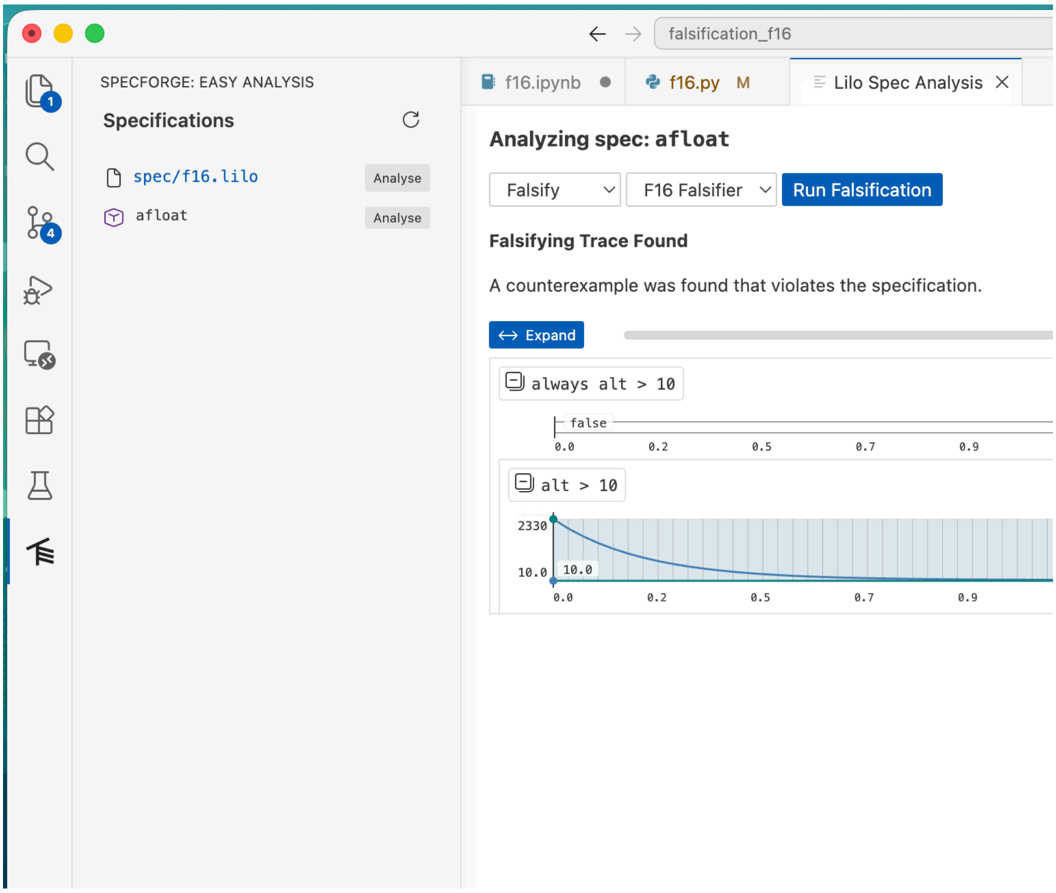
In your `lilo.toml` file, register the falsifier as follows.

```
[[system_falsifier]]
name = "Automatic Transmission Falsifier"
system = "automatic_transmission"
script = "scripts/automatic_transmission.sh"
```

This tells `SpecForge` that the falsifer associated with the system `automatic_transmission` can be invoked by running the script `scripts/automatic_transmission.sh` .

Invoking the Falsification Workflow

Once the Falsification script is ready and registered, it can be invoked using the *easy analysis* option of the VSCode extension. See the screenshot below.



Online Monitoring

SpecForge can be used as an online monitor for Lilo specifications. In contrast with offline monitoring, this means that the monitor will produce verdicts in a streaming manner as it ingests the trace, rather than in one shot after processing the entire trace.

To invoke the online monitor, use the `streammon` command in the `specforge` CLI. Run `specforge streammon --help` for the exact arguments necessary. Running the `specforge streammon` command will start the monitor, which will ingest a trace from STDIN and produce an output signal to STDOUT.

The monitor accepts traces in CSV or JSONL format. In the case of CSV, the monitor expects the first line fed in to be the headers, which can be used to identify the columns corresponding to each signal. In the case of JSONL, the monitor expects each line to be a JSON object with keys corresponding to signal names. The user can specify whether records are encoded in `flat` or `nested` format using the `RECORD_ENCODING` subcommand (see the chapter on Data).

The current implementation of the online monitor performs best when the intervals in temporal operators extend to a small window into the future or past. Intervals with unbounded windows, even if past-time, are not supported.

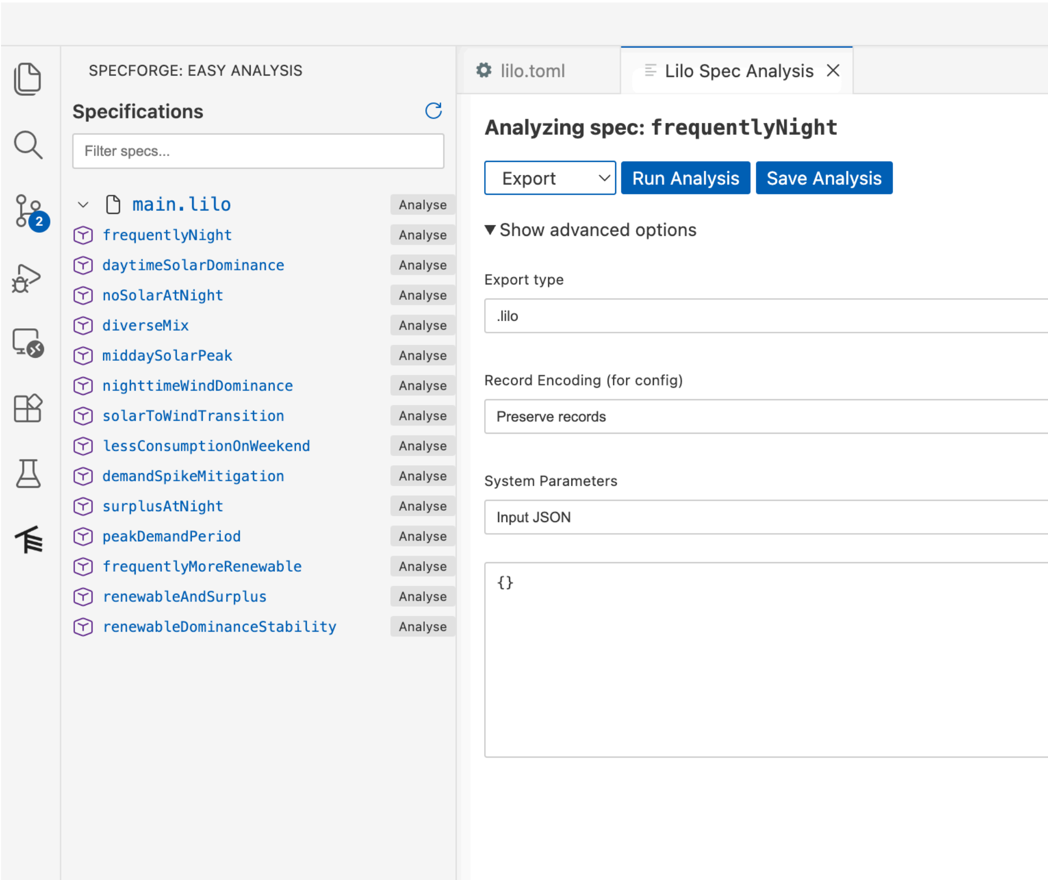
Future-time temporal operators are supported in the online monitor. Their verdicts are produced with a delay corresponding to the largest future window in the spec. For example, when monitoring the formula `eventually [0, 5] f`, the monitor will produce the verdict for the current time `t`, only after ingesting the part of the trace upto time `t + 5`.

Export to RTAMT

SpecForge supports the export of specifications in Lilo to be used by third party monitoring tools. Currently, we only support RTAMT, a monitoring tool for Signal Temporal Logic (STL), maintained by Nickovic et al.

Invoking the Export Functionality

The export functionality can be invoked from the easy analysis UI in the SpecForge VSCode extension, as shown in the screenshot below.



Alternatively, it can also be invoked using the Python SDK for use in a Python Script or a Jupyter Notebook. See the Python SDK documentation for the specifics of this method.

```
rtamt_formula = specforgeClient.export(
    system="f16",
    definition="afloat",
    export_type=EXPORT_RTAMT,
    return_string=True  # Get the exported string
)
print("Exported RTAMT formula:", rtamt_formula)
```

Further Notes on RTAMT

RTAMT is distributed as a Python library. We refer the user to the RTAMT documentation for instructions on how to install and use it.

When exporting to RTAMT, SpecForge will inline definitions to produce a single formula. Parameters will be replaced with their default values, or with values provided as an argument to the export.

Those parameters for which no value is provided will be replaced with a free variable of the same name. If a parameter has a default value, but the user wishes to replace it with a free variable, they can set the parameter value to the JSON value `null`.

The RTAMT formulas involve a much smaller grammar, and therefore the export of Lilo specifications to RTAMT may be limited in some cases. Some of the features of Lilo that are not supported in RTAMT include: usage of strings and records, usage of the `time` variable, usage of non-trivial expressions in intervals, and uses of `if`, `did_change`, `will_change`, `previous_with` and `next_with`. In some cases, the export may still succeed if optimizations applied by SpecForge result in a formula that eliminates the unsupported features.

Sliding window quantitative operations `max_future`, `min_future`, `max_past` and `min_past` are translated to their temporal (`eventually`, `past`, `always` and `historically`) counterparts in RTAMT. This works because RTAMT runs in the robustness mode, and does not typecheck formulas.

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.

Server Connection

- **API Base URL** (`specforge.apiBaseUrl`): The base URL for the backend analysis server. This is ignored if `spawnServer` is enabled. *(string, default: `http://localhost:8080`)*

- **Spawn Server** (`specforge.spawnServer`): If enabled, the extension manages the SpecForge server process locally instead of connecting to an external server via `apiBaseUrl` . *(boolean, default: `false`)*

- **Server Path** (`specforge.serverPath`): The command or absolute path to the SpecForge binary. Requires `spawnServer` to be enabled. *(string, default: `specforge`)*

- **Preferred Port** (`specforge.preferredPort`): The preferred port for the locally spawned SpecForge server. The extension will attempt to use this port, but will fall back to another available port if it is already in use. Requires `spawnServer` to be enabled. *(integer between 1 and 65535, default: `8080`)*

Server Tuning

These settings only apply when `spawnServer` is enabled.

- **Cache Bound** (`specforge.cacheBound`): The size of the LRU cache for SMT solver results. Leave blank for default. *(integer or null, default: `null`)*

- **Semaphore Limit** (`specforge.semaphoreLimit`): The maximum number of concurrent SMT solver processes. Leave blank for default. *(integer or null, default: `null`)*

LLM Integration

These settings configure the AI features used for spec generation and diagnostic explanations. The provider and model settings only apply when `spawnServer` is enabled.

- **LLM Provider** (`specforge.llmProvider`): The LLM provider used for explanations and spec generations. Options: `openai` , `anthropic` , `ollama` , `gemini` , `default` . *(string, default: `default`)*
- **LLM Model** (`specforge.llmModel`): The LLM model used for explanations and spec generations. Leave blank for default. *(string, default: `""`)*
- **OpenAI API Key** (`specforge.openAIApiKey`): The API key for OpenAI. Leave blank to inherit from the environment, or if irrelevant. *(string, default: `""`)*
- **Anthropic API Key** (`specforge.anthropicApiKey`): The API key for Anthropic. Leave blank to inherit from the environment, or if irrelevant. *(string, default: `""`)*
- **Gemini API Key** (`specforge.geminiApiKey`): The API key for Gemini. Leave blank to inherit from the environment, or if irrelevant. *(string, default: `""`)*
- **Ollama API Base** (`specforge.ollamaApiBase`): The API base URL for Ollama. Leave blank to inherit from the environment, or if irrelevant. *(string, default: `""`)*

Other Settings

- **Enable Preview Features** (`specforge.enablePreviewFeatures`): Enable experimental features that are not yet ready for release. *(boolean, default: `false`)*
- **Trace Server** (`specforge.trace.server`): Traces the communication between VS Code and the language server. Useful for debugging extension issues. Options: `off` , `messages` , `verbose` . *(string, default: `off`)*

Initializing a New Project

You can initialize a directory with the recommended project structure from VSCode. Open the command palette (Ctrl+Shift+P / Cmd+Shift+P) and run **SpecForge: Initialize SpecForge Project**. This is equivalent to running the `init` CLI command.

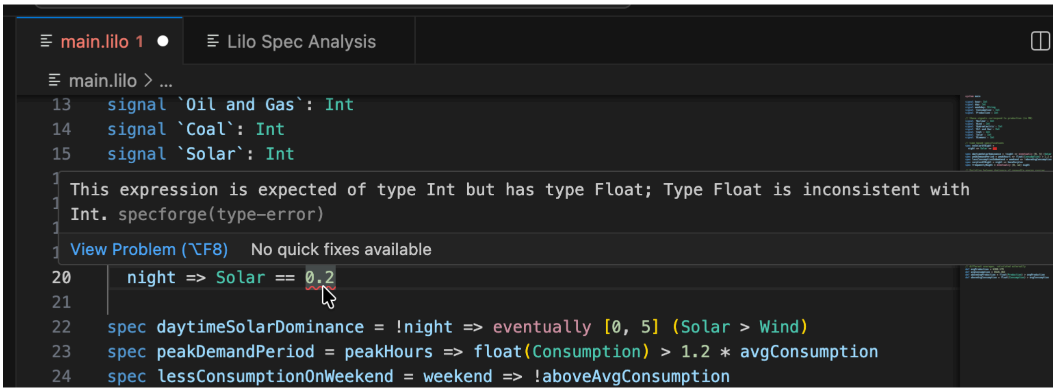
This command will only work if `specforge.spawnServer` is enabled and `specforge.serverPath` is correctly configured, since it relies on the SpecForge binary.

We recommend that when working on a SpecForge project in VSCode, you open the project root directory as the sole workspace folder.

Language Features

Parsing and Type Checking

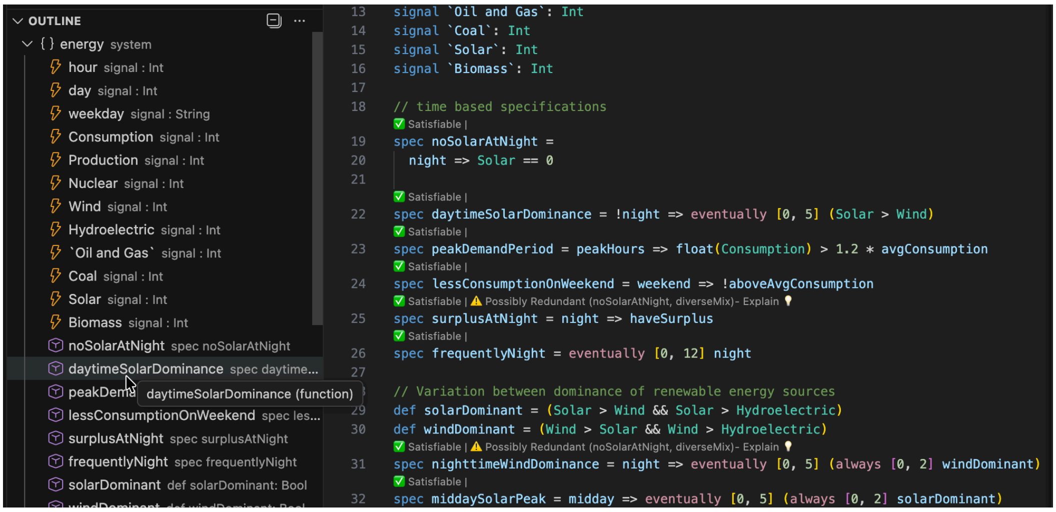
The extension performs real-time checking as you write specifications. Errors will be underlined. Hovering over the affected code will show the error:



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

Document Outline

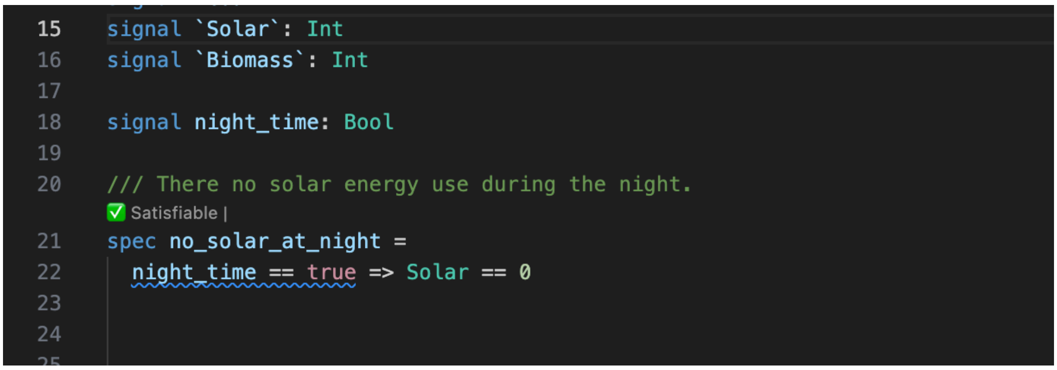
The extension provides a hierarchical outline of your specification file:



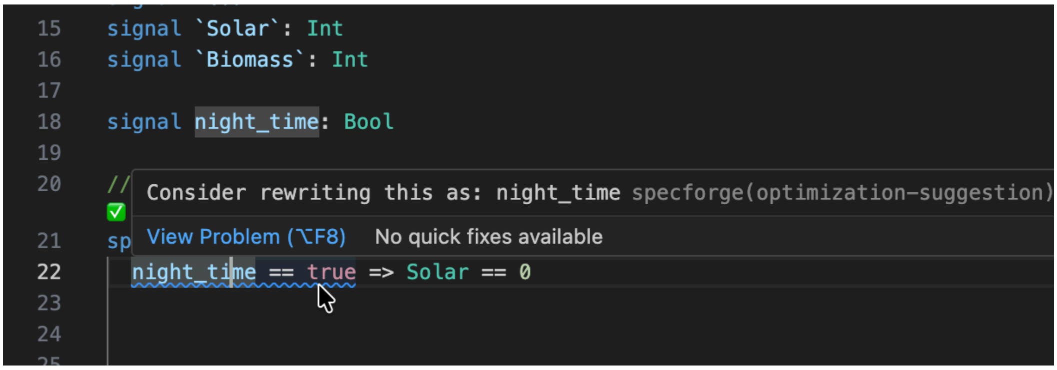
- Open the “Outline” view in VSCode’s Explorer sidebar
- See all specs, definitions, signals, and parameters at a glance
- Click any symbol to jump to its definition
- The outline updates automatically

Diagnostics

The extension performs various checks automatically and provides feedback.



Hovering over a diagnostic will reveal the message.



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:

```
✅ Satisfiable |
spec middaySolarPeak =
  midday => eventually [0, 5] (always [0, 2] solarDominant)
```

When a spec is satisfiable, the "✅ Satisfiable" lens is clickable. Clicking it opens the Spec Analysis pane with the **Exemplify** analysis pre-selected for that spec, so you can immediately generate an example trace that witnesses the spec's satisfiability.

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:

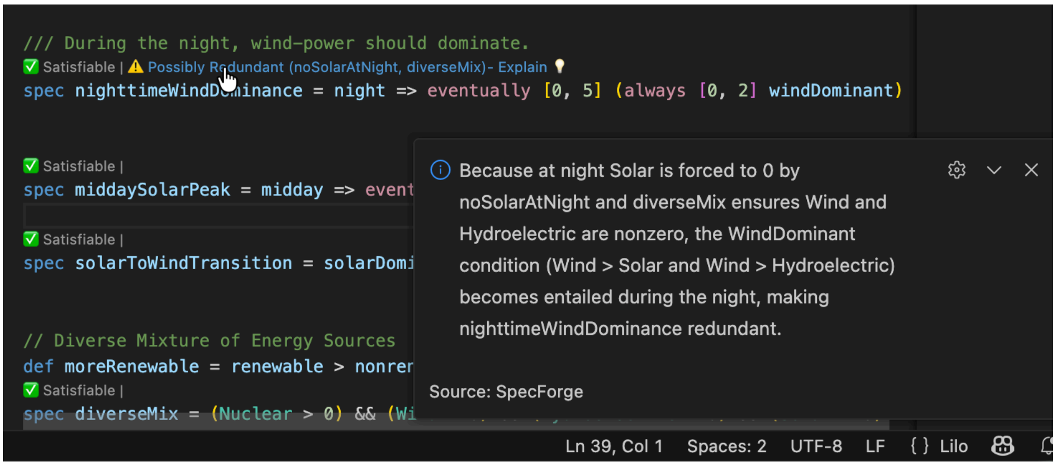
```
def solarDominant = (Solar > Wind && Solar > Hydroelectric)

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

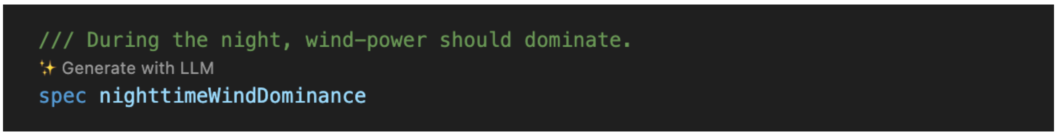
If the system detects that a spec might be redundant, a warning is show as a code lens:

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:



Stubs

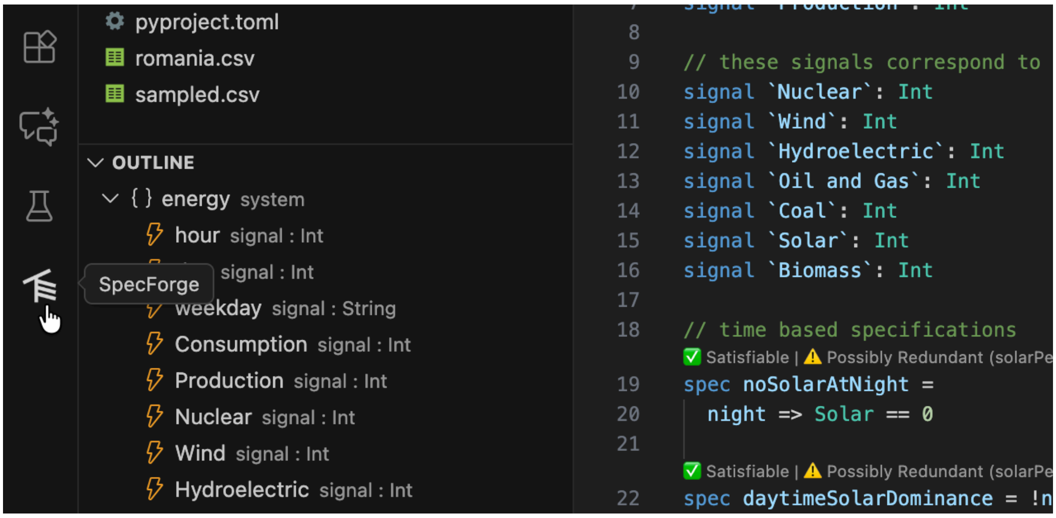
If a `spec` or an `assumption` does not have a body, it is called a *stub*. In this case a code lens offers to generate the specification using AI.



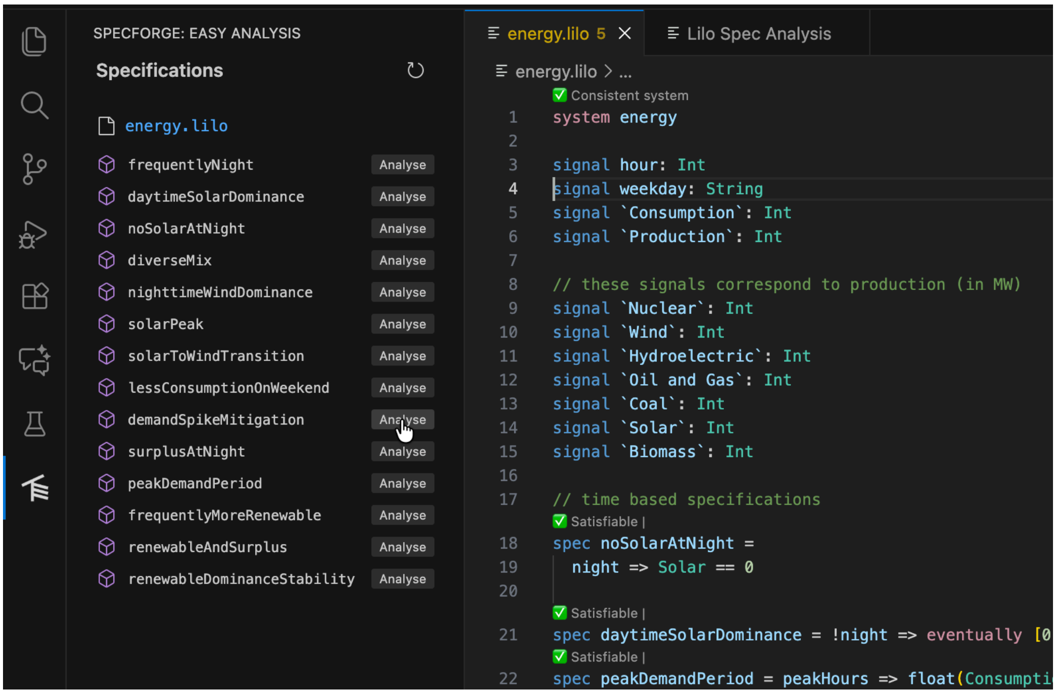
Clicking `Generate with LLM` will produce a definition for the specification, that works with the current system. If the spec is too ambiguous, or if there is some other obstacle to generation, an error message will be shown.

Spec Status Pane

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



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



Clicking `Analyze` 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.

Filtering and Query Syntax

The filter box in the Spec Status Pane can be used to narrow the visible specifications.

Unquoted text is treated as a fuzzy text search, while quoted text must produce an exact match. Both match against:

- the specification name
- docstrings
- aliases

For example, searching for `pump` or `temperature limit` will match specs whose names or metadata are close to those terms.

You can also filter by custom field using `field:value` syntax:

```
priority:>10 reviewed:true owner:"ops"
```

Custom-field predicates only apply to field names that actually exist in the current workspace. Supported comparison operators are: `=`, `≠`, `<`, `≤`, `>`, `≥`.

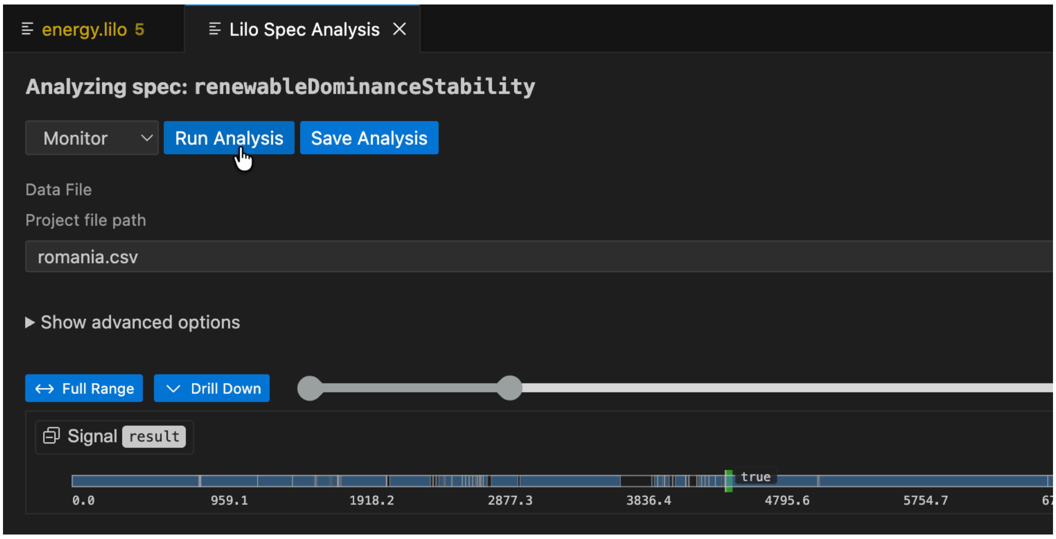
Examples:

- `priority:2`
- `priority:≥2`
- `reviewed:true`
- `owner:"ops"`

Multiple text terms and field predicates are combined with `AND`, so a specification must satisfy all of them to remain visible.

Label chips can also be toggled in the sidebar. Label selections are combined with `OR` across the selected labels, and then combined with the text query as an additional filter.

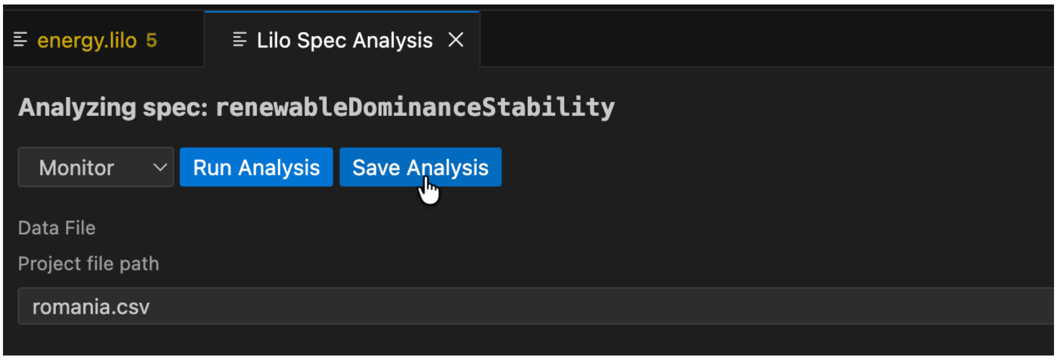
For example, to monitor a specification, select `Monitor` from the dropdown, and choose a data file to monitor, and click `Run Analysis`.



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`.

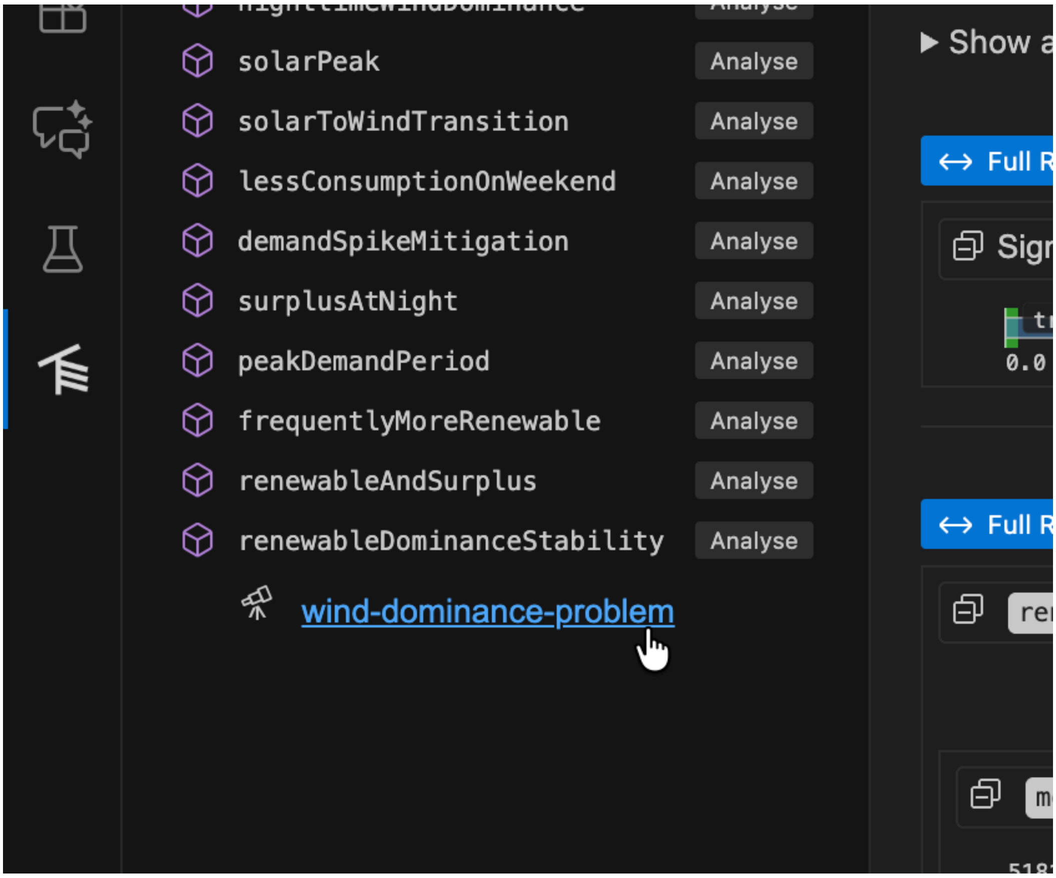
The drill-down has chosen a small enough segment of the full data source in order to present the debugging tree. This will show a tree of monitoring result for the whole specification. Each node can be collapsed or expanded. Hovering on the timeline will also highlight relevant regions in sub-expression 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 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 for the solver to satisfy, each with a rigidity level (Hard or Soft). See Exemplification and Satisfiability for details on assumptions and rigidity levels.

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
- **System Parameters** *(optional)*: Parameter values for the system, provided as a JSON file or inline values. These supply values for `param` declarations in the Lilo system definition.

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 `<cicle>` 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

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:

```python
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 Setting up SpecForge)
- 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

```python
from specforge_sdk import SpecForgeClient

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

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

Core Features

The SDK provides access to core SpecForge capabilities:

- **Monitoring**: Check specifications against data
- **Satisfiability**: Check whether a spec, inline expression, or whole system is satisfiable
- **Validity**: Check whether a spec or expression is provable under the system's assumptions
- **Equivalence**: Check whether two specs or expressions are logically equivalent
- **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

API Methods

`SpecForgeClient(base_url, ...)`

Initialize a SpecForge API client.

Parameters:

- `base_url` (str): The base URL of the SpecForge API server (default: "http://localhost:8080")
- `project_dir` (str/Path): Optional project directory path; if not provided, searches up from current directory for `lilo.toml`
- `timeout` (int): Request timeout in seconds (default: 30)

- `check_version` (bool): Whether to check for version mismatches on initialization (default: True)

Example:

```
specforge = SpecForgeClient(
    base_url="http://localhost:8080",
    project_dir="/path/to/project",
    timeout=60
)
```

`monitor(system, definition, ...)`

Monitor a specification against data. Returns analysis results with verdicts and optional robustness metrics.

Key Parameters:

- `system` (str): The system containing the definition
- `definition` (str | None): Name of the spec to monitor. If `None`, monitors **all** specs in the system at once (system-wide monitoring).
- `data_file` (str/Path): Path to data file (CSV, JSON, or JSONL)
- `data` (list/DataFrame): Direct data as list of dicts or pandas DataFrame
  - **Note:** Provide exactly one of `data_file` or `data`
- `params_file` (str/Path): Path to system parameters file
- `params` (dict): Direct system parameters as dictionary
  - **Note:** Provide at most one of `params_file` or `params`
- `encoding` (dict): Record encoding configuration
- `verdicts` (bool): Include verdict information (default: True)
- `robustness` (bool): Include robustness analysis (default: False)
- `partial_data` (bool): Allow monitoring even when the data does not cover all signals declared in the system (default: True). When enabled, signals that are missing from the data are treated as unknown rather than causing an error. Set to `False` to require that the data provides values for every signal.
- `return_timeseries` (bool): If True, return DataFrame; if False, display JSON (default: False)

When `definition=None`, the monitor evaluates **every** spec in the system against the provided data and returns a combined result. This is useful for getting a quick overview of which specs pass and which fail across the entire system.

Examples:

```python
# Monitor with data file and params file
specforge.monitor(
    system="temperature_sensor",
    definition="always_in_bounds",
    data_file="sensor_data.csv",
    params_file="temperature_config.json"
)

# Monitor with data file and params dict
specforge.monitor(
    system="temperature_sensor",
    definition="always_in_bounds",
    data_file="sensor_data.csv",
    params={"min_temperature": 10.0, "max_temperature": 24.0}
)

# Monitor with DataFrame and get results as DataFrame
result_df = specforge.monitor(
    system="temperature_sensor",
    definition="temperature_in_bounds",
    data=synthetic_df,
    encoding=nested_encoding(),
    params={"min_temperature": 10.0, "max_temperature": 24.0},
    return_timeseries=True
)

# Monitor with robustness analysis
specforge.monitor(
    system="temperature_sensor",
    definition="temperature_in_bounds",
    data_file="sensor_data.csv",
    params={"min_temperature": 10.0, "max_temperature": 24.0},
    robustness=True
)
```

`animate(system, svg_file, ...)`

Create an animation from specification, data, and SVG template.

Key Parameters:

- `system` (str): The system containing the animation definition
- `svg_file` (str/Path): Path to the SVG template file
- `data_file` (str/Path): Path to data file
- `data` (list/DataFrame): Direct data as list of dicts or pandas DataFrame
  - **Note:** Provide exactly one of `data_file` or `data`
- `params_file` (str/Path): Path to a JSON file containing system parameter values
- `params` (dict): Direct system parameters as a dictionary
  - **Note:** Provide at most one of `params_file` or `params`
- `encoding` (dict): Record encoding configuration
- `return_gif` (bool): If True, returns base64-encoded GIF string (default: False)
- `save_gif` (str/Path): Optional path to save the GIF file

Examples:

```python
# Display animation frames in Jupyter
specforge.animate(
    system="temperature_sensor",
    svg_file="temp.svg",
    data_file="sensor_data.csv"
)

# Save animation as GIF file
specforge.animate(
    system="scene",
    svg_file="scene.svg",
    data_file="scene.json",
    save_gif="output.gif"
)

# Get GIF data as base64 string
gif_data = specforge.animate(
    system="temperature_sensor",
    svg_file="temp.svg",
    data=synthetic_df,
    encoding=nested_encoding(),
    return_gif=True
)

# Animate with system parameters
specforge.animate(
    system="temperature_sensor",
    svg_file="temp.svg",
    data_file="sensor_data.csv",
    params={"temp_threshold": 25.0}
)

# Animate with parameters from file
specforge.animate(
    system="temperature_sensor",
    svg_file="temp.svg",
    data_file="sensor_data.csv",
    params_file="temperature_config.json",
    save_gif="output.gif"
)
```

`export(system, definition, ...)`

Export a specification to different formats (e.g., LILO format).

Key Parameters:

- `system` (str): The system containing the definition
- `definition` (str): Name of the spec to export
- `export_type` (dict): Export format configuration (defaults to LILO)
    - Use one of `EXPORT_LILO`, `EXPORT_JSON`, or `EXPORT_RTAMT` defined in the library
- `params_file` (str/Path): Path to system parameters file
- `params` (dict): Direct system parameters as dictionary
    - **Note:** Provide at most one of `params_file` or `params`
- `encoding` (dict): Record encoding configuration
- `return_string` (bool): If True, return exported string; if False, display JSON (default: False)

Examples:

```python
# Export to LILO format as string
lilo_result = specforge.export(
    system="temperature_sensor",
    definition="always_in_bounds",
    export_type=EXPORT_LILO,
    return_string=True
)
print(lilo_result)

# Export with params file
export_result = specforge.export(
    system="temperature_sensor",
    definition="always_in_bounds",
    export_type=EXPORT_LILO,
    params_file="temperature_config.json",
    return_string=True
)

# Export with params dict
export_result = specforge.export(
    system="temperature_sensor",
    definition="always_in_bounds",
    export_type=EXPORT_LILO,
    params={"min_temperature": 10.0, "max_temperature": 24.0},
    return_string=True
)

# Export to JSON format (display in Jupyter)
specforge.export(
    system="temperature_sensor",
    definition="humidity_correlation",
    export_type=EXPORT_JSON
)
```

`check_satisfiability(system, definition=None, expression=None, ...)`

Check satisfiability of a single spec, an inline expression, or the whole system.

Key Parameters:

- `system` (str): The system containing the definition(s)
- `definition` (str | None): Name of the spec to check
- `expression` (str | None): Inline Lilo boolean expression to check in the given system context
  - **Note:** Provide at most one of `definition` or `expression`
- If both `definition` and `expression` are `None`, the SDK checks the whole system
- `timeout` (int | float): Timeout in seconds for the satisfiability query (default: 30)
- `return_details` (bool): If `True`, return the structured satisfiability response instead of a bool (default: `False`)

Returns:

- `bool`: `True` when satisfiable, `False` otherwise.
- If `return_details=True`, returns a dictionary with two keys:
  - `"status"`: The satisfiability outcome. Check `result["status"]["tag"]` for one of:
    - `"Consistent"` – the spec is satisfiable.
    - `"InConsistent"` – the spec is unsatisfiable. The `"contents"` field contains an `UnsatInfo` object with details about the unsat core.
    - `"Unknown"` – the solver could not determine satisfiability.
    - `"TimedOut"` – the solver timed out. The `"contents"` field contains the timeout duration in microseconds.
  - `"param_situation"`: Describes how system parameters (`param` declarations) were handled during the check:
    - `"UsingDefaults"` – the solver fixed all parameters to their declared default values and found the result with those defaults. This is the first strategy attempted.
    - `"UnfixedParams"` – the solver left all parameters free (unconstrained). This happens when the check with default values did not yield a `"Consistent"` result,

so the solver retried with parameters as free variables.

The meaningful combinations of `status` and `param_situation` are:

- (`"Consistent"` , `"UsingDefaults"`) – the spec is satisfiable when the parameters are fixed to their declared defaults.
- (`"Consistent"` , `"UnfixedParams"`) – the spec is satisfiable, but not when the parameters are fixed to their declared defaults.
- (`"InConsistent"` , `"UnfixedParams"`) – for all possible values of the parameters, the spec remains unsatisfiable.

Examples:

```
# Check a single spec
is_satisfiable = specforge.check_satisfiability(
    system="temperature_sensor",
    definition="always_in_bounds"
)
print(is_satisfiable)

# Get the detailed response to inspect the param situation
details = specforge.check_satisfiability(
    system="temperature_sensor",
    definition="always_in_bounds",
    return_details=True
)
print(details["status"]["tag"])        # e.g. "Consistent"
print(details["param_situation"])      # e.g. "UsingDefaults" or "UnfixedParams"

# Check the whole system as a bool
system_is_satisfiable = specforge.check_satisfiability(
    system="temperature_sensor"
)
print(system_is_satisfiable)

# Check an inline expression in the system context
expr_is_satisfiable = specforge.check_satisfiability(
    system="temperature_sensor",
    expression="always (min_temperature ⩽ temperature ⩽ max_temperature)"
)
print(expr_is_satisfiable)
```

`check_validity(system, definition=None, expression=None, ...)`

Check validity of a single spec or an inline expression under the system's assumptions.

Key Parameters:

- `system` (str): The system containing the definition or in whose context the expression is parsed
- `definition` (str | None): Name of the spec to prove
- `expression` (str | None): Inline Lilo boolean expression to prove in the given system context
  - **Note:** Provide exactly one of `definition` or `expression`
- `timeout` (int | float): Timeout in seconds for the validity query (default: 30)
- `return_details` (bool): If `True` , return the structured validity response instead of a bool (default: `False`)

Returns:

- `bool` : `True` when the target is valid, `False` otherwise.
- If `return_details=True` , returns the structured validity result from the SpecForge API. Check `result["status"]["tag"]` for the outcome and `result["param_situation"]` for how defaulted parameters were handled.

Examples:

```python
# Check a single spec
is_valid = specforge.check_validity(
    system="temperature_sensor",
    definition="always_in_bounds"
)
print(is_valid)

# Check an inline expression in the system context
expr_is_valid = specforge.check_validity(
    system="temperature_sensor",
    expression="always (min_temperature ≤ temperature ≤ max_temperature)"
)
print(expr_is_valid)

# Get the detailed response when you need it
details = specforge.check_validity(
    system="temperature_sensor",
    definition="always_in_bounds",
    return_details=True
)
print(details["status"]["tag"], details["param_situation"])
```

`check_equivalence(system, left, right, ...)`

Check whether two specs or inline expressions are logically equivalent under the system's assumptions.

Key Parameters:

- `system` (str): The system in whose context both sides are parsed
- `left` (str): First spec name or inline Lilo expression
- `right` (str): Second spec name or inline Lilo expression
- `timeout` (int | float): Timeout in seconds (default: 30)
- `return_details` (bool): If `True` , return the structured equivalence response instead of a bool (default: `False`)

Returns:

- `bool` : `True` when the two targets are equivalent, `False` otherwise.
- If `return_details=True` , returns the structured equivalence result from the SpecForge API. Check `result["status"]["tag"]` for the outcome and `result["param_situation"]` for how defaulted parameters were handled.

Examples:

```python
# Check that two formulations agree
are_equivalent = specforge.check_equivalence(
    system="temperature_sensor",
    left="always_in_bounds",
    right="always (min_temperature ≤ temperature ≤ max_temperature)"
)
print(are_equivalent)

# Mix of inline expressions
are_equivalent = specforge.check_equivalence(
    system="temperature_sensor",
    left="x > 0 && y > 0",
    right="y > 0 && x > 0"
)
print(are_equivalent)

# Get the detailed response
details = specforge.check_equivalence(
    system="temperature_sensor",
    left="always_in_bounds",
    right="always (min_temperature ≤ temperature ≤ max_temperature)",
    return_details=True
)
print(details["status"]["tag"], details["param_situation"])
```

`exemplify(system, definition, ...)`

Generate example data that satisfies a specification.

Key Parameters:

- `system` (str): The system containing the definition
- `definition` (str): Name of the spec to exemplify
- `assumptions` (list): Additional assumptions to constrain generation (default: []). Each assumption is a dictionary with:
  - `"expression"` (str): A Lilo boolean expression (or the name of an existing spec/def) to assume during generation.
  - `"rigidity"` (str): Either `"Hard"` or `"Soft"` . See Rigidity Levels below.
- `n_points` (int): Number of data points to generate (default: 10)
- `params_file` (str/Path): Path to system parameters file
- `params` (dict): Direct system parameters as dictionary
  - **Note:** Provide at most one of `params_file` or `params`
- `params_encoding` (dict): Record encoding for the parameters
- `timeout` (int): Timeout in seconds for exemplification (default: 30)
- `also_monitor` (bool): Whether to also monitor the generated data (default: True)
- `at_beginning` (bool): If True (the default), the exemplifier searches for a trace whose **first** time point already satisfies the spec. If False, the spec may only become satisfied at a later point in the generated trace. Setting this to `True` is useful when you want the generated example to demonstrate satisfaction from the start; setting it to `False` gives the solver more freedom and can help when the spec involves temporal operators like `past` that are naturally satisfied later in the trace.
- `fix_defaults` (bool): Defaults to `True` . When `True` , the exemplifier is forced to fix any params with default values to those defaults. When `False` , the exemplifier ignores default values and treats params as free variables. See Fixing Params to Default Values for more details.
- `return_timeseries` (bool): If True, return DataFrame; if False, display JSON (default: False)

Rigidity Levels

Each assumption has a rigidity level (`"Hard"` or `"Soft"`) that controls whether the solver treats it as an inviolable constraint or a preference. See Exemplification and Satisfiability for a full explanation.

Examples:

```python
# Generate an example with 20 samples
specforge.exemplify(
    system="temperature_sensor",
    definition="always_in_bounds",
    n_points=20
)

# Generate example with a hard assumption
humidity_assumption = {
    "expression": "eventually (humidity > 25)",
    "rigidity": "Hard"
}
specforge.exemplify(
    system="temperature_sensor",
    definition="humidity_correlation",
    assumptions=[humidity_assumption],
    n_points=20
)

# Mix hard and soft assumptions
specforge.exemplify(
    system="temperature_sensor",
    definition="humidity_correlation",
    assumptions=[
        {"expression": "always_in_bounds", "rigidity": "Hard"},
        {"expression": "eventually (temperature > 30)", "rigidity": "Soft"}
    ],
    params={"min_temperature": 10.0, "max_temperature": 40.0},
    n_points=20
)

# Generate example with partial params (solver fills in missing values)
specforge.exemplify(
    system="temperature_sensor",
    definition="humidity_correlation",
    assumptions=[{"expression": "always_in_bounds", "rigidity": "Hard"}],
    params={"min_temperature": 38.0},
    n_points=20
)

# Allow the spec to be satisfied later in the trace (not necessarily at the first point)
specforge.exemplify(
    system="temperature_sensor",
    definition="recovery_spec",
    n_points=20,
    at_beginning=False
)

# Generate example and get as DataFrame
example_df = specforge.exemplify(
    system="temperature_sensor",
    definition="temperature_in_bounds",
    n_points=15,
    also_monitor=False,
    return_timeseries=True
)
```

`list_defs(system, specs_only=False)`

List all the definitions in the given lilo system.

Parameters:

- `system` (str): The lilo system to list definitions from
- `specs_only` (bool): If True, only list specifications, not other definitions (default: False)

Returns: `list` of `str` - List of definition names

Example:

```python
>>> specforgeClient.list_defs(system ='temperature_sensor', specs_only=True)
['temperature_in_bounds',
 'humidity_correlation',
 'emergency_condition',
 'recovery_spec',
 'always_in_bounds']
```

`health_check()`

Check if the SpecForge API is available and responding.

Returns: `bool` - True if API is healthy, False otherwise

Example:

```
if specforge.health_check():
    print("✓ Connected to SpecForge API")
else:
    print("✗ Cannot connect to SpecForge API")
```

`version()`

Get the API server version and SDK version.

Returns: `dict` with keys `"api"` and `"sdk"`

Example:

```
versions = specforge.version()
print(f"API Version: {versions['api']}")
print(f"SDK Version: {versions['sdk']}")
```

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`

Command Line Interface

The CLI is self-documenting. Run `specforge --help` to see all available commands, and `specforge <command> --help` for detailed usage of any command.

```
$ specforge --help
Usage: specforge COMMAND [--verbose VERBOSITY]

Available commands:
  parse       Check if the Lilo files in a project parse correctly.
  check       Typecheck the Lilo files in a project.
  format      Format the Lilo files in a project.
  monitor     Monitor a spec on a data file, producing an output signal.
  eval        Evaluate a spec at the first timestamp of a data file.
  streammon   Monitor a spec in streaming mode from standard input.
  export      Export a spec to another formalism (Lilo, JSON, RTAMT).
  schema      Generate a schema or data template for a system.
  init        Initialize a new Lilo project.
  flatten     Flatten hierarchical components in a system.
  serve       Start the SpecForge server (for the VS Code extension / Python SDK).
```

Record encoding

Several commands accept a `RECORD_ENCODING` subcommand (`flat` or `nested`) that controls how record-typed signals and parameters are represented. The `flat` encoding accepts `-s` / `--separator SEPARATOR` to set the field separator (default: `"_"`).

When no encoding is specified, the default depends on the file format: **`flat` for CSV**, **`nested` for JSON/JSONL**.

See Record encoding in the Data Files chapter for a full explanation with examples.

Troubleshooting Guide

Running the Doctor

To diagnose issues such as connectivity with the SMT Solver (used for static analysis) or access to LLMs, you can run the `doctor` CLI command. Simply run `specforge doctor --help` to learn more.

Bypassing Proxies

When creating a SpecForge client object using the Python SDK (see the Python SDK guide), make sure that you have specified the correct `base_url` for the server. If you are running the server locally, make sure that you have specified the correct port.

On some systems, there may be a proxy configured which intercepts requests from the Jupyter Notebook (or other Python environments) to the SpecForge server. To ensure that the requests from the Python SDK bypass the proxy, try setting the `NO_PROXY` environment variable to include the address of the SpecForge server (usually `localhost`).

```python
import os

os.environ["NO_PROXY"] = "localhost,127.0.0.1,::1,[::1]"
```

Changelog

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

v0.5.4 - 2025-11-20

Added

- Local signal file monitoring.
- Offline licensing for Docker.
- Monitoring drill-down.
- Monitoring point of interest.
- VSCode Extension documentation.
- Public Docker image available on GHCR with accompanying docs.
- Analysis sidebar now auto-refreshes and shows a spinner while analysis runs.

Change

- Monitor tree sample limit increased to 3100.
- Spec status is no longer a preview feature.

Fixed

- Monitor tree range made responsive.
- Styles of the VSCode sidebar.
- Sample projects for monitoring are bundled correctly.
- Local signal file UX issues resolved.

v0.5.3 - 2025-11-14

Added

- A Whirlwind Tour guide.
- Offline licenses for SpecForge.
- Automated downsampling for monitoring.
- Gemini and Ollama LLM provider support.
- CLI monitoring commands gained interval and sampling options.
- Falsification examples added to the docs.

Fixed

- Issue with stale falsifier list.
- Module name mismatch error reported location.
- CLI commands now run from the project directory.

v0.5.2 - 2025-11-10

Changed

- Falsification timeout from 60 to 240 seconds.

Fixed

- Errors being reported in multiple files.

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 (`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.