

# SpecForgeユーザーガイド

*SpecForge*は、時系列システムを記述するために設計されたドメイン特化言語である*Lilo*をベースにした、AI駆動の形式仕様作成ツールです。

このガイドでは、インストール方法、*Lilo*仕様言語、Python SDK、およびVSCodeでの*Lilo*の使用について説明します。

はじめに、[セットアップ](#)を参照してください。リリースは[リリースページ](#)から入手できます。

このガイドの他のバージョン：

- [PDF形式](#)
- [英語版](#) (Also available [in English](#).)

# SpecForgeのセットアップ

SpecForgeスイートはいくつかのコンポーネントで構成されています：

- **SpecForgeサーバー**：他のコンポーネントが接続するバックエンドサーバーです。Dockerまたは実行ファイルとして実行できます。
- **SpecForge VSCode拡張機能**：VSCodeでLilo言語のサポートを提供し、仕様の編集と管理、およびインタラクティブな可視化のレンダリングを行います。
- **SpecForge Python SDK**：PythonコードからSpecForgeサーバーと対話するためのAPIを提供します。これは、PythonスクリプトやJupyterノートブックからSpecForgeサーバーと仕様やデータをやり取りするために使用できます。

必要なすべてのファイルは、[SpecForgeリリースページ](#)から入手できます。

## クイックスタート

すぐに始めるには、次の手順に従ってください：

1. 依存ツールである `z3` と `rsvg-converter` をインストールする（OS固有の手順を参照; `rsvg-converter` はオプション）
2. お使いのオペレーティングシステム用のSpecForge実行ファイルを[ダウンロード](#)して展開する
3. ライセンスを設定する（お使いのOSに於いて適切な場所に `license.json` を配置）
4. （オプション）環境変数を設定してLLMプロバイダーを設定する（例：  
`SPECFORGE_LLM_PROVIDER=openai`、`OPENAI_API_KEY=...`） - [LLMプロバイダーの設定](#)を参照
5. SpecForgeサーバーを起動する： `./specforge serve`（Windowsでは `.\specforge.exe serve`）
6. [VSCode拡張機能](#)をインストールする（[ドキュメント](#)を参照）
7. プロジェクト用のディレクトリを作成し、`.lilo` ファイルを直接配置する
8. VSCodeでディレクトリを開き、仕様の記述を開始する

**注意：** プロジェクト設定ファイル `lilo.toml` は省略可能です。最初のセットアップでは、これをスキップして、仕様ファイルとデータファイルをプロジェクトルートに直接配置しても差し支えありません。`lilo.toml` をどんな場合にどのように使用するかについては、[プロジェクト設定](#)を参照してください。

## 詳細なセットアップ手順

プラットフォームを選択して詳細なセットアップ手順を確認してください：

- **Windows** - Windowsの完全なセットアップガイド
- **macOS** - macOS（Apple Silicon）の完全なセットアップガイド
- **Linux** - Linuxの完全なセットアップガイド
- **Docker** - 実行ファイルの代わりにDockerを使用する

# WindowsでのSpecForgeのセットアップ

このガイドでは、WindowsでSpecForgeをセットアップする方法を説明します。

## 1. インストール

### MSIインストーラー（推奨）

[SpecForgeリリース](#)ページから `specforge-x.y.z-Windows-X64-ja-JP.msi` をダウンロードし、インストーラーを実行します。

MSIインストーラーは以下を行います：

- SpecForgeを `C:\Program Files\Imiron\SpecForge\` にインストール
- システムのPATHに自動的にSpecForgeを追加
- 必要なすべての依存プログラム（Z3、rsvg-convert）をインストール

### 単一実行ファイル

[SpecForgeリリース](#)ページから `specforge-x.y.z-Windows-X64.zip` をダウンロードし、任意のディレクトリに解凍します。

---

**注意：** この方法では、依存関係を手動でインストールする必要があります。

---

SpecForge実行ファイルには、**Z3**と**rsvg-converter**（オプション）がシステムにインストールされている必要があります。

### Chocolateyの使用（推奨）

PowerShellを管理者権限で開き、次のコマンドを実行します：

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

Chocolateyをお持ちでない場合は、[chocolatey.org](https://chocolatey.org)からインストールできます。

### 手動インストール

パッケージマネージャーを使用したくない場合は、[Z3リリースページ](#)から直接Z3をダウンロードし、PATHに追加してください。

## 2. ライセンスの設定

SpecForgeサーバーを起動するには、有効なライセンスファイルが必要です。ライセンスをお持ちでない場合は、SpecForgeチームにご連絡いただくか、[トライアルライセンス](#)のリクエストをお願い致します。

`license.json` ファイルを次のいずれかの場所に配置します（最初に一致した場所が使用されます）：

### 1. 標準設定ディレクトリ（推奨）：

- `%APPDATA%\specforge\license.json`
- 通常： `C:\Users\YourUsername\AppData\Roaming\specforge\license.json`

2. 環境変数（任意のパスを用いる場合）：

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

3. カレントディレクトリ： `.\license.json`

標準設定ディレクトリが存在しない場合は作成します。PowerShellで以下のように作成できます：

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

## 3. LLMプロバイダーの設定（オプション）

自然言語による仕様生成やエラー説明などのLLMベースの機能を使用するには、サーバーを起動する前に環境変数によってLLMプロバイダーを設定します。

OpenAIの場合（推奨）：

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

APIキーは[platform.openai.com/api-keys](https://platform.openai.com/api-keys)から取得できます。

他のプロバイダー（Gemini、Ollama）については、[LLMプロバイダーの設定ガイド](#)を参照してください。

## 4. サーバーの起動

MSIインストーラーを使用した場合は、任意のディレクトリから以下を実行してSpecForgeを起動できます：

```
specforge serve
```

スタンドアロン実行ファイルを使用した場合は、SpecForge実行ファイルを解凍したディレクトリに移動し、次のコマンドを実行します：

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

サーバーは `http://localhost:8080` で起動します。 <http://localhost:8080/health> にアクセスして、バージョン情報が表示されることを確認できます。

---

**注意：** ライセンスが見つからないか無効な場合、サーバーはすぐに終了します。起動時に問題が発生した場合は、ライセンス設定を確認してください。

---

## 5. VSCode拡張機能のインストール

SpecForge VSCode拡張機能を[Visual Studio Marketplace](#)からインストールするか、[VSCode拡張機能セットアップガイド](#)を参照してください。

## 次のステップ

- [VSCode拡張機能](#) - VSCode拡張機能の機能について学ぶ
- [Python SDK](#) - プログラムによるアクセスのためにPython SDKをセットアップする
- [SpecForge早わかり](#) - SpecForgeの機能を体験する
- [プロジェクト設定](#) - `lilo.toml` による設定について学ぶ

# macOSでのSpecForgeのセットアップ

このガイドでは、スタンドアロン実行ファイルを使用してmacOSでSpecForgeをセットアップする手順を説明します。

## 1. 実行ファイルのダウンロード

SpecForgeリリースページから `specforge-x.y.z-macOS-ARM64.tar.bz2` をダウンロードし、任意のディレクトリに展開します。

## 2. 依存関係のインストール

SpecForge実行ファイルを動かすには、**Z3**がインストールされている必要があります。Homebrewを使用してインストールします：

```
brew install z3
```

Homebrewがインストールされていない場合は、[brew.sh](https://brew.sh)からインストールしてください。

## 3. ライセンスの設定

SpecForgeサーバーを起動するには、有効なライセンスファイルが必要です。ライセンスをお持ちでない場合は、SpecForgeチームにお問い合わせいただくか、[試用ライセンス](#)をリクエストしてください。

`license.json` ファイルを以下のいずれかの場所に配置します（最初に見つかったものが使用されます）：

1. 標準設定ディレクトリ（推奨）：

- `~/.config/specforge/license.json`

2. 環境変数（任意のパスを用いる場合）：

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

3. カレントディレクトリ： `./license.json`

標準設定ディレクトリが存在しない場合は作成します：

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

## 4. LLMプロバイダーの設定（オプション）

自然言語による仕様生成やエラー説明などのLLMベースの機能を使用するには、サーバーを起動する前に環境変数によってLLMプロバイダーを設定します。

OpenAIの場合（推奨）：

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

APIキーは[platform.openai.com/api-keys](https://platform.openai.com/api-keys)から取得してください。

その他のプロバイダー（Gemini、Ollama）については、[LLMプロバイダー設定ガイド](#)を参照してください。

## 5. サーバーの起動

SpecForge実行ファイルを展開したディレクトリに移動し、次のコマンドを実行します：

```
./specforge serve
```

サーバーは <http://localhost:8080> で起動します。 <http://localhost:8080/health> にアクセスして、バージョン情報が表示されることを確認できます。

---

**注意：**ライセンスが見つからないか無効な場合、サーバーは直ちに終了します。起動時に問題が発生した場合は、ライセンスの設定を確認してください。

---

### ダウンロードしたSpecForgeバイナリの実行許可

ダウンロードしたバイナリはサードパーティのソースからダウンロードされたものであるため、[MacOS Gatekeeper](#)がバイナリの実行を阻止する警告を表示する場合があります。

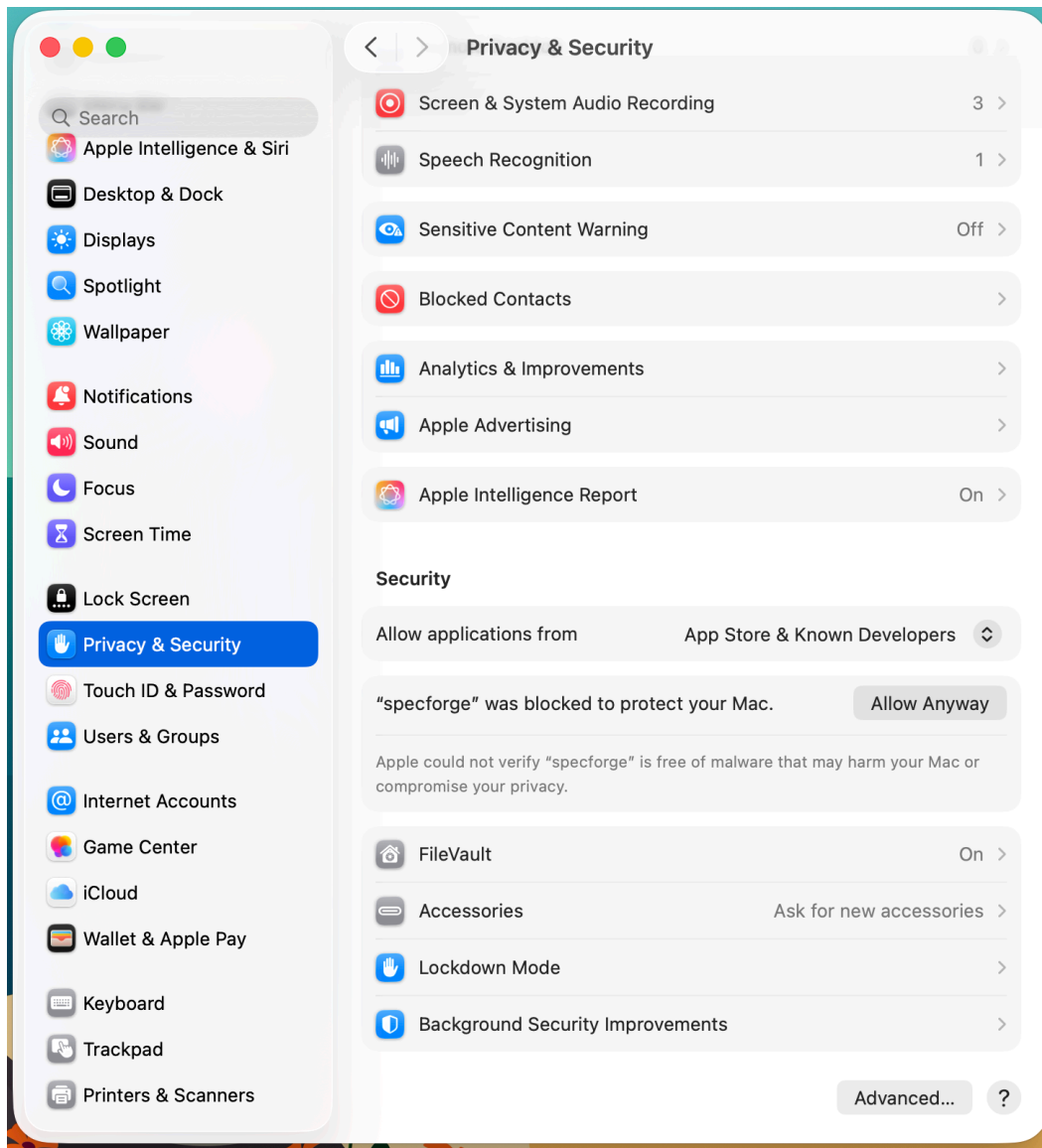


specforge実行ファイルをホワイトリストに追加するには、次のコマンドを実行します。

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

または、システム設定のGUIから次の手順で実行することもできます。

1. システム設定を開き、「プライバシーとセキュリティ」に移動します
2. セキュリティセクションに、「"specforge"はMacを保護するためにブロックされました。」と表示されるはず
3. 「このまま開く」をクリックします。



## 6. VSCode拡張機能のインストール

[Visual Studio Marketplace](#)からSpecForge VSCode拡張機能をインストールするか、[VSCode拡張機能のセットアップガイド](#)を参照してください。

### 次のステップ

- [VSCode拡張機能](#) - VSCode拡張機能の機能について学ぶ
- [Python SDK](#) - プログラムによるアクセスのためにPython SDKをセットアップする
- [SpecForge早わかり](#) - SpecForgeの機能を体験する
- [プロジェクト設定](#) - `lilo.toml` による設定について学ぶ



# LinuxでのSpecForgeのセットアップ

このガイドでは、スタンドアロン実行ファイルを使用してLinuxでSpecForgeをセットアップする方法を説明します。

## 1. 実行ファイルのダウンロード

[SpecForgeリリースページ](#)から `specforge-x.y.z-Linux-X64.tar.bz2` をダウンロードし、任意のディレクトリに解凍します。

## 2. 依存関係のインストール

SpecForge実行ファイルが動作するには、**Z3**がインストールされている必要があります。お手元のディストリビューションのパッケージマネージャーを使用してインストールしてください：

Ubuntu/Debian：

```
sudo apt install z3
```

Fedora/RHEL：

```
sudo dnf install z3
```

Arch Linux：

```
sudo pacman -S z3
```

## 3. ライセンスの設定

SpecForgeサーバーを起動するには、有効なライセンスファイルが必要です。ライセンスをお持ちでない場合は、SpecForgeチームにご連絡いただくか、[トライアルライセンス](#)のリクエストをお願い致します。

`license.json` ファイルを次のいずれかの場所に配置します（最初に一致した場所が使用されます）：

1. **標準設定ディレクトリ**（推奨）：

- `~/.config/specforge/license.json`

2. **環境変数**（任意のパスを用いる場合）：

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

3. **カレントディレクトリ**：`./license.json`

標準設定ディレクトリが存在しない場合は作成します：

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

## 4. LLMプロバイダーの設定（オプション）

自然言語による仕様生成やエラー説明などのLLMベースの機能を使用するには、サーバーを起動する前に環境変数によってLLMプロバイダーを設定します。

OpenAIの場合（推奨）：

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

APIキーは[platform.openai.com/api-keys](https://platform.openai.com/api-keys)から取得できます。

他のプロバイダー（Gemini、Ollama）については、[LLMプロバイダーの設定ガイド](#)を参照してください。

## 5. サーバーの起動

SpecForge実行ファイルを解凍したディレクトリに移動し、次のコマンドを実行します：

```
./specforge serve
```

サーバーは <http://localhost:8080> で起動します。 <http://localhost:8080/health> にアクセスして、バージョン情報が表示されることを確認できます。

---

**注意：** ライセンスが見つからないか無効な場合、サーバーはすぐに終了します。起動時に問題が発生した場合は、ライセンス設定を確認してください。

---

## 6. VSCode拡張機能のインストール

SpecForge VSCode拡張機能を[Visual Studio Marketplace](#)からインストールするか、[VSCode拡張機能セットアップガイド](#)を参照してください。

## 次のステップ

- [VSCode拡張機能](#) - VSCode拡張機能の機能について学ぶ
- [Python SDK](#) - プログラムによるアクセスのためにPython SDKをセットアップする
- [SpecForge早わかり](#) - SpecForgeの機能を体験する
- [プロジェクト設定](#) - `lilo.toml` による設定について学ぶ

# DockerでのSpecForgeのセットアップ

このガイドでは、スタンドアロン実行ファイルの代わりにDockerを使用してSpecForgeをセットアップする方法を説明します。

## 前提条件

サーバーを起動する前に、有効なライセンスを取得して設定する必要があります。ライセンスをお持ちでない場合は、SpecForgeチームにご連絡いただくか、[トライアルライセンス](#)のリクエストをお願い致します。

## 1. Docker Composeファイルの取得

SpecForgeサーバーは、GHCR（GitHub Container Registry）を介してDockerイメージとして配布されています。Dockerイメージを実行する方法としては、Docker Composeを用いての実行を推奨します。

最新の `docker-compose-x.y.z.yml` ファイルを[SpecForgeリリースページ](#)からダウンロードしてください。

## 2. ライセンスの設定

SpecForgeサーバーには有効なライセンスファイルが必要です。ライセンスファイルをDockerコンテナで利用可能にする必要があります。

1. `license.json` ファイルをホストマシン上の既知の場所に配置します（例：  
`~/.config/specforge/license.json`）。
2. `docker-compose-x.y.z.yml` ファイルの次の行を変更して、ライセンスファイルを指すようにします：

```
- type: bind
  source: ~/.config/specforge/license.json # このパスがライセンスファイルを指すようにしてください
  target: /app/license.json # このパスは変更しないでください
  read_only: true
```

---

**注意：** 多くの場合、`docker`を `root` として、つまり `sudo` を使用して実行することが一般的です。その場合、`~/` で始まるパスはシステムによって `/root/` として解釈されます。混乱を避けるために、絶対パスを使用するか、`docker`を `root` として実行しないようにすることができます。

---

## 3. LLMプロバイダーの設定（オプション）

自然言語による仕様生成やエラー説明などのLLMベースの機能を使用するには、サーバーを起動する前に `docker-compose.yml` ファイルで環境変数を変更してLLMプロバイダーを設定します。

OpenAIの場合（推奨）：

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

APIキーは[platform.openai.com/api-keys](https://platform.openai.com/api-keys)から取得できます。

APIキーを直接ファイルに挿入することもできますが、セキュリティのために環境変数を使用する方が推奨されます。

他のプロバイダー（Gemini、Ollama）および詳細な設定オプションについては、[LLMプロバイダーの設定ガイド](#)を参照してください。

## 4. サーバーの起動

次のコマンドを実行します。 `/path/to/docker-compose-x.y.z.yml` を、ダウンロードしたファイルへの実際のパスに置き換えてください：

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

---

起動時にエラーが生じた場合にコンテナがすみやかに異常終了できるようにするために、 `--abort-on-container-exit` フラグを付加しておくことを推奨します。

---

サーバーが起動していることを確認するには、 <http://localhost:8080/health> にアクセスして、バージョン情報が表示されることを確認してください。

---

**注意：** ライセンスが見つからないか無効な場合、サーバーはすぐに終了します。起動時に問題が発生した場合は、ライセンス設定を確認してください。

---

## 5. VSCode拡張機能のインストール

SpecForge VSCode拡張機能を[Visual Studio Marketplace](#)からインストールするか、[VSCode拡張機能セッティングガイド](#)を参照してください。

## Dockerイメージの更新

SpecForgeサーバーの新しいバージョンが定期的にリリースされます。最新バージョンは次のいずれかの方法で使用できます：

1. [リリースページ](#)から更新されたdocker-composeファイルを使用する、または
2. Docker Composeファイルのimageフィールドをlatestに設定する：

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

その後、最新のイメージをプルします：

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

## 次のステップ

- [VSCode拡張機能](#) - VSCode拡張機能の機能について学ぶ
- [Python SDK](#) - プログラムによるアクセスのためにPython SDKをセットアップする
- [SpecForge早わかり](#) - SpecForgeの機能を体験する
- [プロジェクト設定](#) - `lilo.toml` による設定について学ぶ

# LLMプロバイダーの設定

SpecForgeには、自然言語による仕様生成やエラー説明などのLLMベースの機能が含まれています。これらの機能を使用するには、LLMプロバイダーを設定する必要があります。

## サポートされているプロバイダー

SpecForgeは現在、3つのLLMプロバイダーをサポートしています：

- **OpenAI** - クラウドベースのAPI（ほとんどのユーザーに推奨）
- **Gemini** - GoogleのクラウドベースのAPI
- **Ollama** - マシン上でモデルをローカルで実行

## 設定方法

### 実行ファイル用（Windows、macOS、Linux）

SpecForgeサーバーを起動する前に、次の環境変数を設定します：

#### 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"
```

APIキーは[platform.openai.com/api-keys](https://platform.openai.com/api-keys)から取得できます。

#### 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"
```

APIキーは[ai.google.dev/gemini-api/docs/api-key](https://ai.google.dev/gemini-api/docs/api-key)から取得できます。

#### Ollama

まず、[docs.ollama.com/quickstart](https://docs.ollama.com/quickstart)からOllamaをインストールして実行します。

次に、環境変数を設定します：

```
# Linux / macOS
export SPECFORGE_LLM_PROVIDER=ollama
export SPECFORGE_LLM_MODEL=your-model-name # 例: 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" # 例: llama3.2、mistral
$env:OLLAMA_API_BASE="http://127.0.0.1:11434"
```

Ollamaサーバーが別のマシンで実行されている場合は、OLLAMA\_API\_BASE を変更してください。

## Docker用

docker-compose.yml ファイルの環境変数を変更します：

```
- SPECFORGE_LLM_PROVIDER=openai # その他のオプション: ollama、gemini
- SPECFORGE_LLM_MODEL=gpt-5-nano-2025-08-07 # プロバイダーに適したモデルを選択
# LLM_PROVIDERに応じて次のいずれか:
- OPENAI_API_KEY=${OPENAI_API_KEY}
- GEMINI_API_KEY=${GEMINI_API_KEY}
- OLLAMA_API_BASE=http://127.0.0.1:11434 # ollamaサーバーがリモートで実行されている場合は変更してください
```

APIキーを直接ファイルに挿入することもできます：

```
- SPECFORGE_LLM_PROVIDER=gemini
- GEMINI_API_KEY=abc123XYZ # 文字列の引用符は不要
```

ただし、セキュリティのために環境変数を使用の方が推奨されます。

## デフォルトモデル

SPECFORGE\_LLM\_MODEL 変数を設定しない場合：

- **OpenAI**：デフォルトは gpt-5-nano-2025-08-07
- **Gemini**：デフォルトは gemini-2.5-flash
- **Ollama**：モデルを指定する必要があります（デフォルトなし）

## LLM設定なしの場合

適切なLLMプロバイダー設定がない場合、LLMベースのSpecForge機能は利用できません。SpecForgeの残りの部分は通常どおり動作し続けます。

# VSCode拡張機能

VSCode用のLilo言語拡張機能は以下を提供します：

- .lilo ファイルのシンタックスハイライト、型検査、入力補完
- 仕様の充足可能性と冗長性のチェック
- Pythonノートブックでのモニタリング結果の可視化のサポート

## インストール

SpecForge VSCode拡張機能は2つの方法でインストールできます：

### VSCode Marketplaceから

拡張機能を[Visual Studio Marketplace](#)から直接インストールするか、VSCodeの拡張機能タブ（Ctrl+Shift+X または Cmd+Shift+X）で「SpecForge」を検索します。

### VSIXファイルから

または、VSIXファイル（[リリース](#)に含まれています）からインストールできます：

- VSCodeの拡張機能タブを開き（Ctrl+Shift+X または Cmd+Shift+X）、右上の3つのドットをクリックして、`Install from VSIX...`を選択します
- VSCodeのコマンドパレットを開き（Ctrl+Shift+P または Cmd+Shift+P）、`Extensions: Install from VSIX...`と入力して、`.vsix` ファイルを選択します

---

**重要：** 拡張機能のバージョンがSpecForgeサーバーのバージョンと一致していることを確認してください。バージョンの不一致は互換性の問題を引き起こす可能性があります。

---

## 使用方法と設定

- 拡張機能が動作するには、SpecForgeサーバーが実行されている必要があります（[SpecForgeのセットアップ](#)を参照）。
- 必要に応じて、[拡張機能設定](#)を使用してサーバーに対応するURIを設定できます。このURLの末尾にはスラッシュを追加しないでください。

拡張機能がインストールされ、サーバーが実行されている状態で、.lilo ファイルおよび関連するPythonノートブックで自動的に動作するようになります。



# Python SDKのセットアップ

Python SDKは、ノートブックを含むPythonプログラム内からSpecForgeツールとプログラマ的に対話するために使用できるPythonライブラリです。

Python SDKは、 `specforge_sdk-x.x.x-py3-none-any.whl` という名前のwheelファイルとしてパッケージ化されています。

SDKの機能と能力の概要については、[SpecForge Python SDKガイド](#)を参照してください。

## サンプルウォークスルー

Python SDKは、 `pip` を使用して直接インストールするか、 `poetry` や `uv` などのビルド環境を介して依存関係として定義できます。

以下では、 `uv` を使用してこのような環境をセットアップする方法について説明します。別のビルドシステムを使用する場合も、ワークフローは同様です。

1. お使いのオペレーティングシステムに `uv` をインストールします。詳細については、[uvインストールガイド](#)を参照してください。
2. 新しいプロジェクトディレクトリを作成して移動します。 `pyproject.toml` ファイルを配置します。
3. `pyproject.toml` ファイルで依存関係を宣言します。
  - Python SDK用のwheelファイルは、ローカル依存関係として宣言できます。wheelファイルへの正しいパスが提供されていることを確認してください。
  - インタラクティブモニターなどのSpecForgeの機能は、Python Notebookの一部として使用できます。そのためには、 `jupyterlab` も依存関係として含めることをお勧めします。
  - `numpy`、 `pandas`、 `matplotlib` などのライブラリは、データ処理と視覚化のために頻繁に含まれます。
  - 以下は `pyproject.toml` ファイルの例です：

```
[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-0.5.6-py3-none-any.whl" }
```

4. `uv sync` を実行します。これにより、適切な依存関係（正しいバージョンのpythonを含む）がインストールされた `.venv` ディレクトリが作成されます。
5. `source .venv/bin/activate` を実行して、 `python` にアクセスできるShell Hookを使用します。これが正しく構成されていることは、以下のように確認できます。

```
$ source .venv/bin/activate
(falsification-examples) $ which python
/path/to/project/falsification/.venv/bin/python
```

6. これで、サンプルノートブックを閲覧できます。ノートブックが `.venv` 内のカーネルに接続されていることを確認してください。これは通常自動的に構成されますが、手動でも実行できます。手動で行うには、`jupyter server` を実行し、VSCode ノートブックビューアーのカーネル設定でサーバーURLをコピー＆ペーストします。

# プロジェクト設定

Liloプロジェクトには、プロジェクトルートに**任意で** `lilo.toml` という設定ファイルを配置することができます。

初めて使う場合はこの設定をスキップして、**.lilo 仕様ファイルとデータファイルをプロジェクトのルートに直接配置するだけで構いません**。SpecForgeは適切なデフォルト値で動作します。

`lilo.toml` を使用する場合、ファイル全体または一部フィールドが欠落していても、適切なデフォルト値が適用されます。Python SDKとVS Code拡張機能は、このファイルを読み取り、それに応じてセマンティクスを適用します。

設定ファイルは以下のような場合に有用です：

- プロジェクト名とカスタムソースパスの設定（デフォルト: プロジェクトルートまたは `src/`）
- 言語の動作のカスタマイズ（インターバルモード、フリーズ）
- 診断設定の調整（整合性、冗長性、最適化、未使用の定義）とそのタイムアウト
- 反証解析のための `system_falsifier` エントリの登録

以下に、スキーマとデフォルト値、その後に完全な例を示します。

## スキーマとデフォルト値

トップレベルのキーと、省略された場合のデフォルト値:

- `project`
  - `name` (文字列).
    - デフォルト: `""`。
    - 初期化時: 指定された名前に設定されます。指定がない場合は、プロジェクトルートディレクトリの名前になります。
  - `source` (パス文字列). デフォルト: `"src/"`
- `language`
  - `interval.mode` (文字列). サポート: `"static"`. デフォルト: `"static"`
  - `freeze.enabled` (ブール値). デフォルト: `true`
- `diagnostics`
  - `consistency.enabled` (ブール値). デフォルト: `true`
  - `consistency.timeouts.named` (秒、浮動小数点数). デフォルト: `0.5`
  - `consistency.timeouts.system` (秒、浮動小数点数). デフォルト: `1.0`
  - `redundancy.enabled` (ブール値). デフォルト: `true`
  - `redundancy.timeouts.named` (秒、浮動小数点数). デフォルト: `0.5`
  - `redundancy.timeouts.system` (秒、浮動小数点数). デフォルト: `1.0`
  - `optimize.enabled` (ブール値). デフォルト: `true`
  - `unused_defs.enabled` (ブール値). デフォルト: `true`
- `[[system_falsifier]]` (テーブルの配列、オプション)
  - 各エントリ: `name` (文字列)、`system` (文字列)、`script` (文字列)
  - 存在しないか空の場合、このキーはファイルから省略され、空のリストとして扱われます

デフォルトファイル:

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

## lilo.tomlの例

オーバーライドを含むプロジェクトの例。

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

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

[diagnostics.consistency]
enabled = true

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

[diagnostics.optimize]
enabled = true

[diagnostics.redundancy]
enabled = false

[diagnostics.unused_defs]
enabled = false

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

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

# SpecForge早わかり

このセクションは、実際の例を通してSpecForgeの主な機能を素早く紹介します。Lilo言語で仕様を書く方法と、SpecForgeのVSCode拡張機能を使用して解析する方法を探ります。

## Lilo言語：概要

Liloは、ハイブリッドシステム向けに設計された式ベースの時相仕様言語です。主な概念は次のとおりです：

**基本型**：Bool、Int、Float、およびString

**演算子**：標準的な算術演算子（+、-、\*、/）、比較演算子（==、<、> など）、および論理演算子（&&、||、=>）

**時相演算子**：Liloの特徴的な機能は、豊富な時相論理演算子のセットです：

- `always  $\phi$` ： $\phi$  がすべての未来の時刻で真である
- `eventually  $\phi$` ： $\phi$  がある未来の時刻で真である
- `past  $\phi$` ： $\phi$  がある過去の時刻で真であった
- `historically  $\phi$` ： $\phi$  がすべての過去の時刻で真であった

これらの演算子は時間間隔で修飾することができます。例えば、`eventually[0, 10]  $\phi$`  は、 $\phi$  が10時間単位以内に真になることを意味します。その他の演算子は[こちら](#)をご覧ください。

**システム**：Lilo仕様はシステムに編成され、以下をグループ化します：

- `signal`：時間変動する入力値（例：`signal temperature: Float`）
- `param`：時間変動しない非時間パラメータ（例：`param max_temp: Float`）
- `type`：構造化データのためのカスタム型
- `def`：再利用可能な定義とヘルパー関数
- `spec`：システムが満たすべき要件

システムファイルは `system temperature_control` のようなシステム宣言で始まり、そのシステムのすべての宣言が含まれます。

言語の包括的なガイドについては、[Lilo言語](#)の章を参照してください。

## 実行例

温度制御システムを実行例として使用します。このサンプルプロジェクトは、[リリース](#)で入手できます。このシステムは温度と湿度のセンサーを監視し、値が安全な範囲内に保たれることを保証する仕様を持っています：

```

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

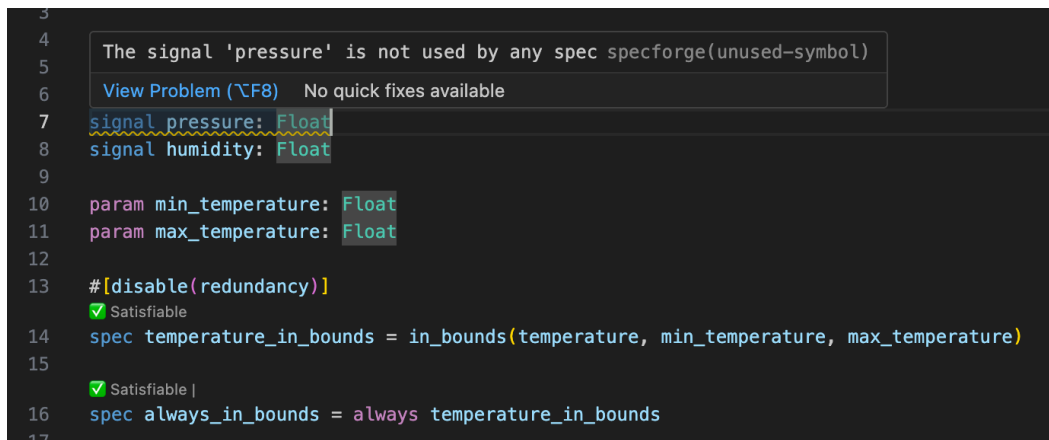
// Humidity should be reasonable when temperature is in normal range
spec humidity_correlation = always (
  (temperature >= 15.0 && temperature <= 35.0) =>
  (humidity >= 20.0 && humidity <= 80.0)
)

// 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)
)

```

VSCode拡張機能は、Liloコードの記述、構文強調表示、型検査、警告、仕様の充足可能性などをサポートします：



## 仕様の解析

システムの仕様を記述したら、SpecForge VSCode拡張機能はさまざまな解析機能を提供します：

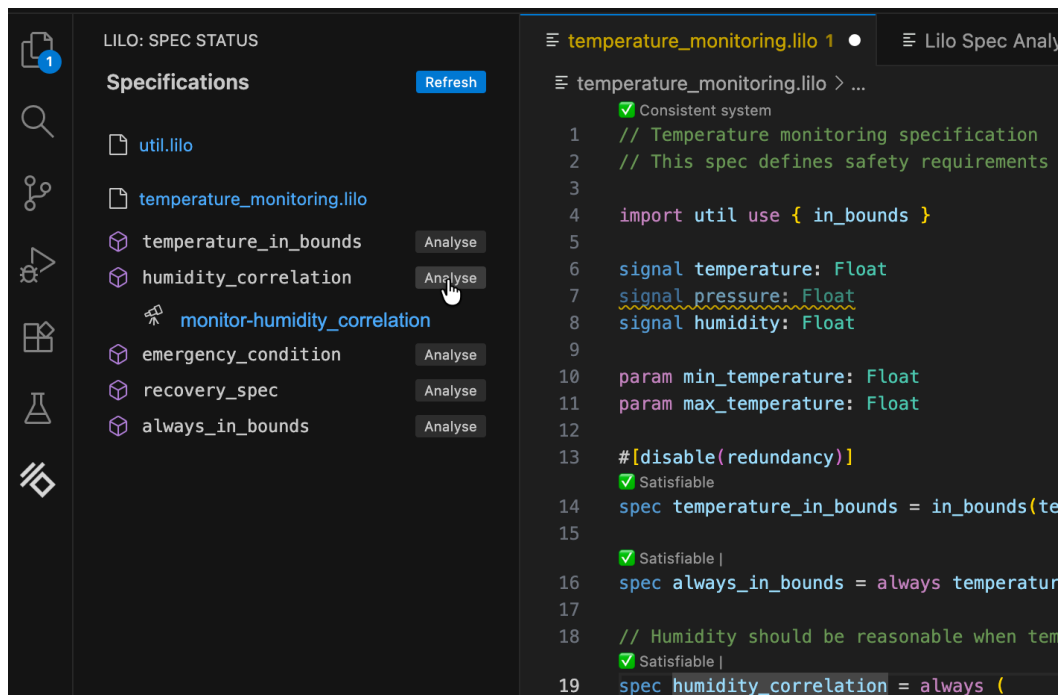
- **Monitor**：記録されたシステムの動作が仕様を満たしているかチェック
- **Exemplify**：仕様を満たす例のトレースを生成
- **Falsify**：モデルに対して、仕様に違反する反例を検索
- **Export**：仕様を他の形式（.json、.lilo など）に変換
- **Animate**：仕様の動作を時間経過とともに視覚化

これはVSCode内から直接実行することも、[Python SDK](#)を使用してJupyterノートブック内から実行することもできます。ここではVSCode内で直接解析を実行します。[VSCodeガイド](#)では、すべての機能についてさらに詳しく説明しています。

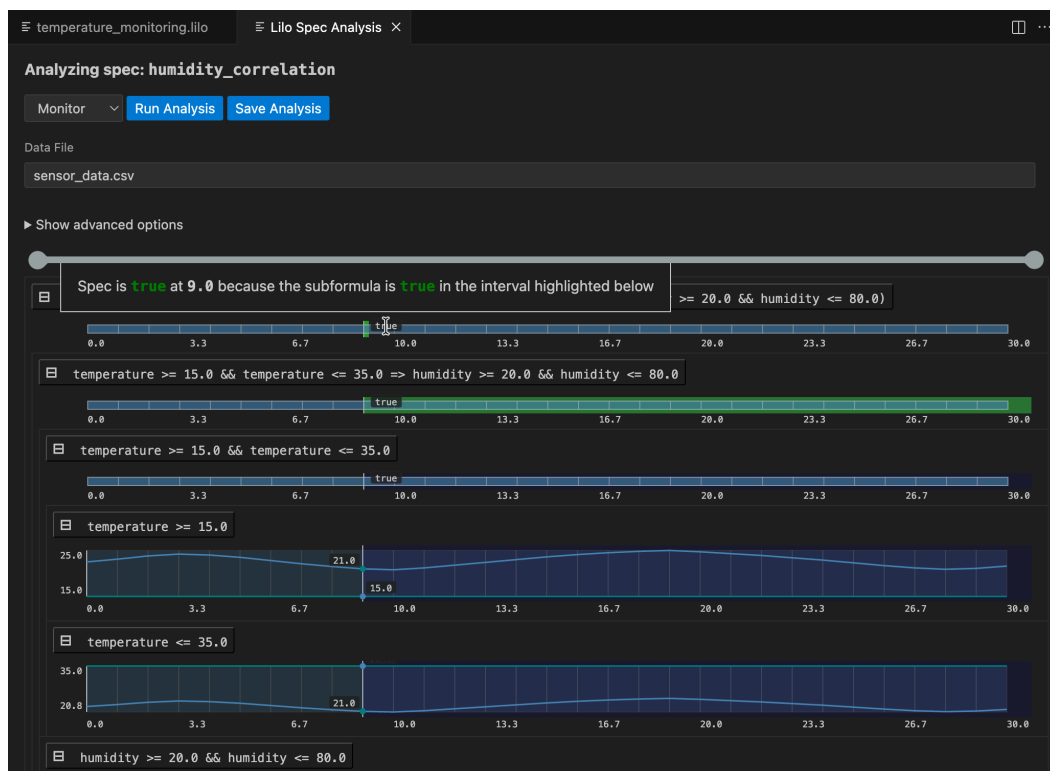
## モニタリング

モニタリングは、データファイルに記録された実際のシステムの動作が仕様を満たしているかをチェックします。記録されたトレースデータを提供すると、SpecForgeがそれに対して仕様を評価します。

仕様選択画面に移動し、モニタリングしたい仕様の **Analyse** ボタンをクリックします。

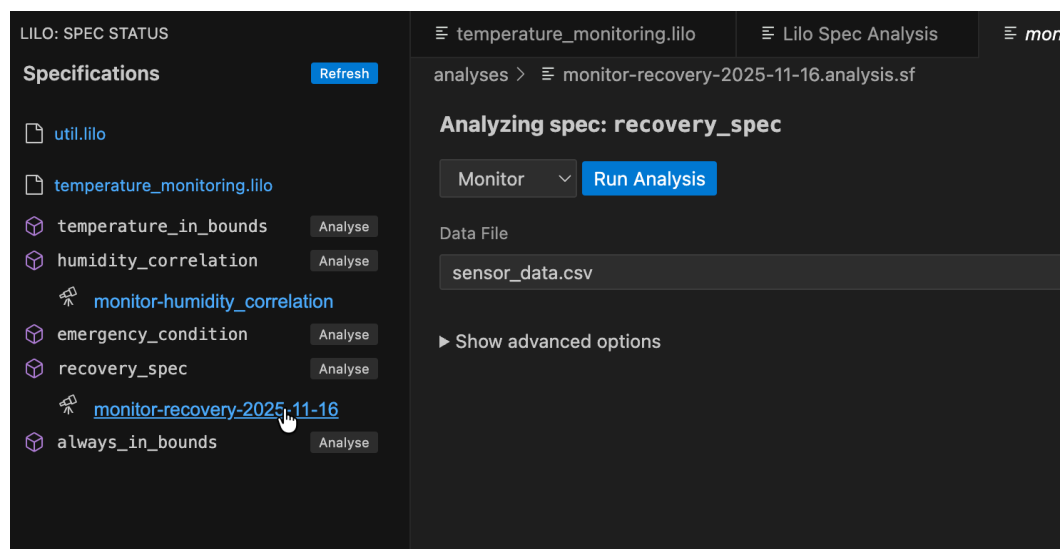


ドロップダウンメニューからデータファイルを選択した後、**Run Analysis** をクリックします。結果は仕様の解析モニタリングツリーです：



仕様全体の結果が上部に表示されます。その下では、仕様のサブ式を掘り下げて、任意の時点で仕様が真または偽になる理由を理解できます。任意のシグナルにカーソルを合わせると、その時点での結果の説明を含むポップアップが表示され、サブ式の結果シグナルの関連するセグメントが強調表示されます。

解析は保存できます。保存するには、Save Analysis ボタンをクリックし、解析を保存する場所を選択します。その後、この解析ファイルに移動して、VSCodeで再度開くことができます。解析は、関連する仕様の下にある仕様ステータスメニューにも表示されます。



## 実証化

Exemplify 解析は、満足する動作を示す例のトレースを生成します。これは以下のような場合に便利です：

- 有効なシステムの動作がどのようなものかを理解する
- リアルなデータで他のコンポーネントをテストする
- アニメーションを作成する



実証化されたデータが期待どおりに動作しない場合、仕様が間違っている可能性があり、修正が必要です。実証化は、仕様を作成する際の補助として使用できます。



## 反証化

システムのモデルが利用可能な場合、反証化を使用して、モデルが期待どおり、つまり仕様に従って動作するかどうかを確認できます。

最初に、lilo.toml に反証器を登録する必要があります。例：

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

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

これが完了すると、反証器が Falsify 解析メニューに表示されます。反証化シグナルが見つかった場合、モニタリングツリーが表示され、モデルがどこで間違ったかを理解するのに役立ちます：



## エクスポート

Export は、仕様を他のツールで使用するために他の形式に変換します。たとえば、仕様をJSON形式にエクスポートする場合は、Export type として .json を選択します。

Analyzing spec: humidity\_correlation

Export

Run Analysis

Save Analysis

▼ Show advanced options

Export type

Record Encoding

.json

Preserve records

Record Encoding (for config)

Preserve records

System Parameters

Input.JSON

{}

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

## 次のステップ

このツアーでは、SpecForgeができることの基本をカバーしました。以下の章では、より詳しく掘り下げます：

- 完全なLilo言語 ([Lilo言語](#))
- システムの定義と構成 ([システム](#))
- プログラムによるアクセスのためのPython SDK ([Python SDK](#))

# Lilo言語

Liloは、複雑な時間依存システムの動作を記述、検証、監視するために設計された形式仕様言語です。

Liloにより以下のことができます：

- 時間軸にまたがる性質を定義するための強力な時相演算子を備えつつも馴染みのある構文を使用して式を記述できます。
- レコードを使用してシステムのデータをモデル化するためのデータ構造を定義できます。
- システムと呼ばれる機構を使用して仕様を構造化できます。

## 型と式

Liloは式ベースの言語です。つまり、単純な算術から複雑な時相プロパティまで、ほとんどの構造は時系列値に評価される式です。このセクションでは、Lilo式の基本的な構成要素について詳しく説明します。

### コメント

コメントブロックは `/*` で始まり、`*/` で終わります。これらのマーカー間のすべては無視されます。

行コメントは `//` で始まり、行の残りがコメントであることを示します。

ドキュメント文字列は `//` の代わりに `///` で始まり、さまざまな言語要素にドキュメントを添付します。

### プリミティブ型

Liloは型付き言語です。プリミティブ型は次のとおりです：

- `Bool`：ブール値。`true` と `false` が該当します。
- `Int`：整数値。例：`42`
- `Float`：浮動小数点値。例：`42.3`
- `String`：テキスト文字列。二重引用符で囲まれて書かれます。例：`"hello world"`

型エラーは次のようにユーザーに通知されます：

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

このドキュメントページのコードブロックは編集可能です。例えば、型エラーを修正するために `n` の型を `Float` に変更してみてください。

### 測定単位

Liloは物理単位付きの `Float` 値をサポートしています。単位はリテラルの直後に山括弧 `<...>` を使って記述します。

#### 基本単位

単純な単位は山括弧内に識別子として記述されます：

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

#### 複合単位

単位は演算子を使って組み合わせることができます。リテラル `1` は無次元の単位を表します：

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

- 比 (`/`)：

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

- 積 ( \* ) :

50.0<m\*m>

- べき乗 ( ^ ) :

100.0<m^2>

9.81<m\*s^-2>

## 演算子の優先順位と結合性

単位演算子は標準的な数学的優先順位規則に従います：

1. べき乗 ( ^ ) は最も優先順位が高く、直前の単位に強く結合します。
2. 積 ( \* ) と 比 ( / ) は同じ優先順位を持ち、左から右に結合します。

したがって、 $m/s*kg$  は  $(m/s)*kg$  として解釈され、また  $m*s^{-2}$  は  $m*(s^{-2})$  であって  $(m*s)^{-2}$  ではありません。

## グループ化のための括弧

括弧を使用して結合の優先順位を明示できます：

1.0<1/(kg\*m)>

## 演算子

Liloは以下の演算子を使用します。優先順位の高い順（最高から最低）にリストされています。

- 前置否定： $\neg x$  は  $x$  の符号反転、 $!x$  は  $x$  の否定です。
- 乗算と除算： $x * y$ 、 $x / y$
- 加算と減算： $x + y$ 、 $x - y$
- 数値比較：
  - $==$  : 等しい
  - $!=$  : 等しくない
  - $>=$  : 以上
  - $<=$  : 以下
  - $>$  : 真に大きい
  - $<$  : 真に小さい 比較は同方向のものを連鎖させることができます。例えば  $0 < x <= 10$  は  $0 < x \ \&\& \ x <= 10$  と同じ意味です。
- 時相演算子
  - $\text{always } \phi$  :  $\phi$  は将来の全ての時点で真
  - $\text{eventually } \phi$  :  $\phi$  は将来のある時点で真
  - $\text{past } \phi$  :  $\phi$  は過去のある時点で真だった
  - $\text{historically } \phi$  :  $\phi$  は過去の全ての時点で真だった
  - $\text{will\_change } \phi$  :  $\phi$  は将来のある時点で値が変わる
  - $\text{did\_change } \phi$  :  $\phi$  は過去のある時点で値が変わった
  - $\phi \text{ since } \psi$  :  $\psi$  が過去のある時点で真で、それ以前の全ての時点で  $\phi$  が真だった
  - $\phi \text{ until } \psi$  :  $\psi$  が真になるまでの将来の全ての時点で  $\phi$  が真
  - $\text{next } \phi$  :  $\phi$  は1つ次の（離散的な）時点で真
  - $\text{previous } \phi$  :  $\phi$  は1つ前の（離散的な）時点で真

時相演算子は区間で修飾できます：

- `always [a, b]  $\phi$`  : `a` 時間単位先から `b` 時間単位先までの間の全ての時点で  $\phi$  が真
  - `eventually [a, b]  $\phi$`  : `a` 時間単位先から `b` 時間単位先までの間のある時点で  $\phi$  が真
  - `$\phi$  until [a, b]  $\psi$`  : `a` 時間単位先から `b` 時間単位先までの間のある時点で  $\psi$  が真になり、それまでの全ての時点で  $\phi$  が真
  - 同様の区間修飾が他の時相演算子でも使えます。
  - 区間で `infinity` を使用できます： `[0, infinity]`。
- 連言 `x && y` : `x` と `y` の両方が真
  - 選言 `x || y` : `x` または `y` のいずれかが真
  - 含意と等価：
    - `x => y` : `x` が真の場合、`y` も真でなければならない
    - `x <=> y` : `x` が真であるのは、`y` が真であるとき、かつそのときに限る

前置演算子は連鎖できないことに注意してください。したがって、`-( $\neg$ x)` や `!(next  $\phi$ )` のように書く必要があります。

## 組み込み関数

いくつかの組み込み関数が用意されています：

- `float` は `Int` から `Float` を生成します：

```
def n: Int = 42

def x: Float = float(n)
```

- `time` はシグナルの現在時刻を返します。

## 条件分岐

条件分岐を使用すると、特定の式の真偽に基づいて仕様を異なる値へと評価させることができます。 `if - then - else` 構文を使用します。

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

Liloの重要な機能は、 `if / then / else` が文ではなく**式**であることです。つまり、常に値に評価されるため、 `else` 分岐は必須です。

`if` 直後の式は `Bool` に評価される必要があります。 `then` 節と `else` 節は互換性のある型の値を生成する必要があります。例えば、 `then` 節が `Int` に評価される場合、 `else` 節も `Int` に評価される必要があります。

条件分岐は式が期待される場所ならどこでも使用でき、より複雑なロジックを処理するためにネストできます。

```
// ゼロ除算を回避
def safe_ratio(numerator: Float, denominator: Float): Float =
  if denominator != 0.0 then
    numerator / denominator
  else
    0.0 // デフォルト値を返す

// ネストされた条件式
def describe_temp(temp: Float): String =
  if temp > 30.0
  then "hot"
  else if temp < 10.0
  then "cold"
  else
    "moderate"
```

`if _ then _ else _` は時点ごと(pointwise)であることに注意してください。つまり、条件はすべての時点に独立して適用されます。

## レコード

レコードは、フィールドと呼ばれる名前付きの値をグループ化する複合データ型で、仕様内で構造化データをモデル化するために不可欠です。

Lilo言語は、匿名の、構造的に型付けされた、拡張可能なレコードをサポートしています。

### 構築と型

中括弧で囲まれた `field = value` ペアのカンマ区切りリストを提供することで、レコード値を構築できます。レコードの型は、フィールド名とそれらに対応する値の型から推論されます。

例えば、次の式は2つのフィールドを持つレコードを作成します：型 `Int` の `foo` と型 `String` の `bar`。

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

結果の値は構造的型 `{ foo: Int, bar: String }` を持ちます。コンストラクタ内のフィールドは順不同です。

`type` 宣言を使用して名前付きレコード型を宣言することもできます。明確性と再利用性の観点から、レコード型は `type` 宣言で定義することを強く推奨します。

```
/// 2D座標系の点を表します。
type Point = { x: Float, y: Float }

// Point型の値を構築
def origin: Point = { x = 0.0, y = 0.0 }
```

### フィールドパニング

レコードに同名で格納されるべき名前がすでにスコープ内にある場合、明示的な割り当てを省略してフィールドをパン(pun)できます。例えば `{ foo }` は `{ foo = foo }` の省略形です。

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

def record_with_puns = { foo, bar }
```

パニングは、レコードリテラルや更新を含む、レコードフィールドがリストされる場所ならどこでも機能します。各パンは、型チェック中に通常の `field = value` ペアに展開されます。

## パスフィールド構築

ネストされたレコードは、ドット区切りのパスで各フィールドを記述することで、一度に作成または拡張できます。最後のフィールドに至るまでの各セグメントはその断片を含むレコードを指すものであり、コンパイラは各断片をレコード式へと適切に統合してくれます。

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

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

フィールドはやはり順不同で、どのような順序で記述されていても適切なレコードにマージされます。ドット区切りのパスはパニングと組み合わせることはできません。ドット区切りのパスで指定する場合は、`{ status.throttle }`の代わりに`{ status.throttle = throttle }`と書きます。

## withによるレコード更新

`{ base with fields }`を使用して、既存のレコードをコピーし、特定のフィールドを上書きします。更新は、レコード構築と同じ構文規則に従います：通常の割り当て、パン、ドット付きパスを混在させることができます。

```
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 }
```

更新されたすべてのフィールドは、ベースレコードにすでに存在する必要があります。パス更新を使用すると、構造全体を再構築することなく、深くネストされた部分を書き換えることができます。

## 射影

レコード内のフィールドの値にアクセスするには、ドット（.）構文を使用します。p が x という名前のフィールドを持つレコードの場合、`p.x` はこの値にアクセスする式です。

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

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

レコードはネストでき、射影は連鎖できます。

```
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
```

## ローカルバインディング

ローカルバインディングを使用すると、式に名前を割り当てることができ、その後の式で使用できます。これは `let` キーワードを使用して実現され、仕様の明確性、構造、効率を向上させるために非常に貴重です。

ローカルバインディングは `let name = expression1; expression2` という形式をとります。これは `expression1` の結果を `name` にバインドします。バインディング `name` は、バインディングのスコープである `expression2` 内でのみ表示されます。



let バインディングの主な目的は次のとおりです：

1. **可読性**：複雑な式をより小さな名前付きの部分に分割することで、ロジックを理解しやすくします。
2. **再利用**：部分式が複数回使用される場合、それを名前にバインドすることで繰り返しと潜在的な再計算を回避できます。

辺の長さ  $a$ 、 $b$ 、 $c$  から三角形の外接円の面積を計算する次の式を考えてみましょう：

```
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))
```

let バインディングを使用すると、ロジックがはるかに明確になります：

```
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
```

バインドされた変数 ( $s$ 、 $area$ 、 $circumradius$ ) の型は、それが割り当てられた式から自動的に推論されます。複数の let バインディングを連鎖させて、計算を段階的に構築することもできます。

## システム

最終的に、Liloはシステムを指定するために使用されます。システムは、時系列入力シグナル、（非時系列の）パラメータ、および仕様の宣言をまとめたものです。システムには補助的な定義も含まれます。

システムファイルは、システム宣言で始まる必要があります。例：

```
system Engine
```

システムの名前はファイル名と一致する必要があります。

## 型宣言

新しい型は `type` キーワードで宣言されます。例えば、新しいレコード型 `Point` を定義するには以下のよう記述します：

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

その後、ファイル内の他の場所で `Point` を型として使用できます。

## シグナル

システムの時間変化する値はシグナルと呼ばれます。これらは `signal` キーワードで宣言されます。例：

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

システムの定義と仕様は、システムのシグナルを自由に参照できます。

シグナルは、関数型を含んでいない任意の型、つまりプリミティブ型とレコードの組み合わせにすることができます。

## システムパラメータ

時間的に定数であるシステムの変数はシステムパラメータと呼ばれ、`param` キーワードで宣言されます。例：

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

システムの定義と仕様は、システムのパラメータを自由に参照できます。システムパラメータは、監視 (monitoring) を開始する前に事前に提供が必要があることに注意してください。一方で、例示 (exemplification) に際してはシステムパラメータの値は省略できます。つまり、指定された場合、例はそれらに準拠して作成され、指定されなかった場合、例示のプロセスはそのシステムパラメータとして有効な値を見つけようとします。

## 定義

定義は `def` キーワードで宣言されます：

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

定義はパラメータに依存することができます：

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

定義の戻り値の型を指定することもできます：

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

パラメータの型アノテーションと戻り値の型は両方とも省略可能で、指定されなかった場合は推論されます。ただし、これらの型註釈はある種のドキュメントとして常にかいておくことをお勧めします。

定義のパラメータもレコード型にすることができます。例：

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

定義は他の定義で使用できます。例：

```
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)
```

定義は、循環依存を作成しない限り、任意の順序で指定できます。

定義は、パラメータとして宣言することなく、システムの任意のシグナルを自由に使用できます。

## 仕様

仕様は、システムについて真であるべきことを記述したもので、システムのすべての `signal` と `def` を参照でき、`spec` キーワードを使用して宣言されます。 `def` によく似ていますが、次の点が異なります：

- 戻り値の型は常に `Bool` です（指定する必要はありません）
- パラメータを持つことはできません。

例：

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

## モジュール

Lilo言語はモジュールをサポートしています。モジュールはモジュール宣言で始まり、(システムと同様に) いくつかの定義から成っています：

```
module Util

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

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

モジュールの名前はファイル名と一致する必要があります。例えば、`module Util` として宣言されたモジュールは、`Util.lilo` という名前のファイルで定義する必要があります。

モジュールには `def` と `type` のみを含めることができます。

他のモジュールからアクセス可能にしたい定義は、`pub` (「public」の意) を付与する必要があります。

モジュールを使用するには、例えば `import Util` のようにインポートする必要があります。その後、`Util` の `pub` な定義が修飾名で使えるようになります。例：

```
import Util

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

モジュールをエイリアスで修飾してインポートすることもできます。例：

```
import Util as U

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

修飾子なしでシンボルを使用するには、`use` キーワードを使用します：

```
import Util use { calc }

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

## 静的解析

システムコードはいくつかのコード品質チェックを通過します。

### 一貫性チェック

仕様は一貫性がチェックされます。仕様が充足不可能かもしれない場合、警告が生成されます：

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

これは、どのシステムもこの仕様を満たすことができないため、仕様に問題があることを意味します。

仕様間の不整合もユーザーに報告されます：

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

この場合、各仕様は単独では充足可能ですが、いかなるシステムもこれら両方ともを満たすことはできません。

### 冗長性チェック

ある仕様が、システムの他の仕様によって暗黙的に示されているために冗長である場合、これも検出されます：

```
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
```

この場合、`sometimes_negative` という仕様が冗長であることをユーザーに警告します。なぜなら、このプロパティは `positive_becomes_negative` と `sometimes_positive` の組み合わせによって既に暗示されているためです。実際、`sometimes_positive` は  $x > 0$  となる時点が存在することを意味し、`positive_becomes_negative` に基づけば、その時点以後に  $x < 0$  となる時点が存在しなければならないと結論付けられます。

# 追加機能

## 属性

Liloの定義、仕様、パラメータ、シグナルは、`///` で始まるdocstringに加え、属性でも注釈を付けることができます。属性は、注釈対象の直前に配置する必要があります。

属性は、ツール（特にVSCode拡張機能）によって使用される、注釈付き項目に関するメタデータを伝えるために使用されます。

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

- 未使用変数の警告を抑制する： `#[disable(unused)]`
  - この属性を定義、パラメータ、またはシグナルに使用すると、未使用であることに関する警告が抑制されます。
  - 仕様または公開されている定義は、常に使用されているとみなされます。
- 静的解析のタイムアウト
  - 静的解析のデフォルトのタイムアウトをオーバーライドするには、`timeout` を秒単位で指定できます。
  - 次のようにして静的解析の項目ごとのタイムアウトを個別に指定できます： `#[timeout(satisfiability = 20, redundancy = 30)]`
  - または、次のようにして両方を10秒に設定することもできます： `#[timeout(10)]`
- 静的解析を無効にする
  - `#[disable(satisfiability)]` または `#[disable(redundancy)]` を使用して、定義に対する特定の静的解析を無効にします。

## パラメータのデフォルト値

システムを記述する際、パラメータにはデフォルト値を指定することもできます。こうしたデフォルト値は、典型的な用例ではパラメータがそのデフォルト値でインスタンス化されるであろうことを意図するために使えます。

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

デフォルト値は定数を宣言するために使用すべきものではありません。定数には、代わりに `def` を使用してください。

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

- モニタリング時、デフォルト値を持つパラメータは入力から省略できます。省略された場合、デフォルト値が使用されます。明示的に指定することもでき、その場合は指定された値が使用されます。
- 式をエクスポートする際、デフォルト値を持つパラメータは、エクスポート前にデフォルト値で置換されます。
- デフォルト値が設定されている場合、Exemplification（例示化）機能は、ソルバーにパラメータをデフォルト値に固定するよう要求します（デフォルト値が与えられていない場合は、パラメータは自由変数として扱われます）。

エクスポートや例示化などの分析を実行する際、設定のフィールドにJSONの `null` を指定することができます、これによりSpecForgeは該当パラメータのデフォルト値を無視するようになります。

```
system main

signal p: Int
param bound: Int = 1

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

- 例示化 (Exemplification)
  - `params = {}` の場合：充足不可能 ( `bound` のデフォルト値が使用されます )
  - `params = { "bound": null }` の場合：充足可能 (ソルバーは制約を満たす `bound` の値を自由に選択できます)
- エクスポート
  - `params = {}` の場合：出力結果は `p > 1 && p < 1`
  - `params = { "bound": 100 }` の場合：出力結果は `p > 1 && p < 100`
  - `params = { "bound": null }` の場合：出力結果は `p > 1 && p < bound`

JSONの `null` 自体をLiloプログラム中でデフォルト値として使用することはできません。

## 仕様スタブ

ユーザーは、実体を持たない仕様（仕様スタブ）を定義できます。こうしたスタブも、通常の設定と同様にdocstringと属性を持つことができます。スタブはプレースホルダーとして使用でき、Liloの周辺ツールによって `true` として解釈されます。

VSCode拡張機能は、（設定されている場合は）docstringに基づいてLLMによりスタブの実装を生成するためのコードレンズを表示します。

```
/// システムは常に最終的にエラーから回復する必要があります。
spec error_recovery
```

# 規約

一部の言語では、名前のクラスを区別するために、特定の名前を大文字始めにする必要があったりしますが、Liloは命名に関して柔軟性があり、仕様を記述するシステムの命名規則に合わせることが出来ます。とはいえ、本ドキュメント内の例では以下の規約に基づいた名前を使用します：

- モジュールとシステムの名前は小文字のみからなるスネークケースです。例えば、ClimateControlではなく climate\_control とします。
- **重要:** モジュールまたはシステムの名前は、それが定義されているファイル名と一致する必要があります。例えば、module climate\_control または system climate\_control は、climate\_control.lilo という名前のファイルで定義する必要があります。
- signal、param、def、spec の名前、引数、レコードフィールド名は小文字のみからなるスネークケースです。例えば、signal WindSpeed や signal windSpeed ではなく signal wind\_speed とします。
- ユーザー定義のものも含め、型の名前は大文字で始まるキャメルケースにする必要があります：

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



# VSCode拡張機能

SpecForge VSCode拡張機能は、Lilo仕様を記述および解析するための包括的な開発環境を提供します。言語サポート、対話型解析ツール、および視覚化機能が、統一されたインターフェースで繋がっています。

セットアップについては、[VSCode拡張機能のインストールガイド](#)を参照してください。

## 概要

この拡張機能は以下を提供します：

- **言語サポート**：Language Server Protocol (LSP) 実装による構文強調表示、型検査、およびオートコンプリート
- **診断**：型エラー、未使用の定義、および最適化提案のリアルタイムチェック
- **コードレンズ**：コードに直接埋め込まれた対話型解析ツール
- **仕様ステータスペイン**：仕様と保存された解析をナビゲートするための専用サイドバー
- **仕様解析ペイン**：仕様のモニタリング、実証化、反証化などのための対話型GUI
- **ノートブック統合**：JupyterノートブックでのSpecForge結果の視覚化のサポート
- **LLM機能**：AI駆動の仕様生成と診断の説明

## 設定

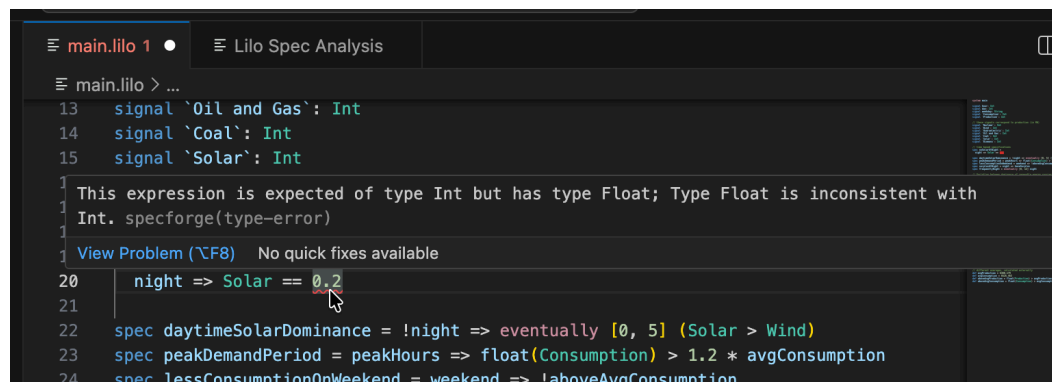
拡張機能には、実行中のSpecForgeサーバーが必要です。VSCodeの設定でサーバー接続を構成します：

- **API Base URL**：SpecForgeサーバーのURI（デフォルト： `http://localhost:8080` ）
  - 設定 → 拡張機能 → SpecForge → Api Base Url からアクセス
  - または設定ID： `specforge.apiBaseUrl` を使用
- **Enable Preview Features**：仕様ステータスサイドバーを含む実験的機能を有効化
  - 設定 → 拡張機能 → SpecForge → Enable Preview Features からアクセス
  - または設定ID： `specforge.enablePreviewFeatures` を使用

## 言語機能

### 解析と型検査

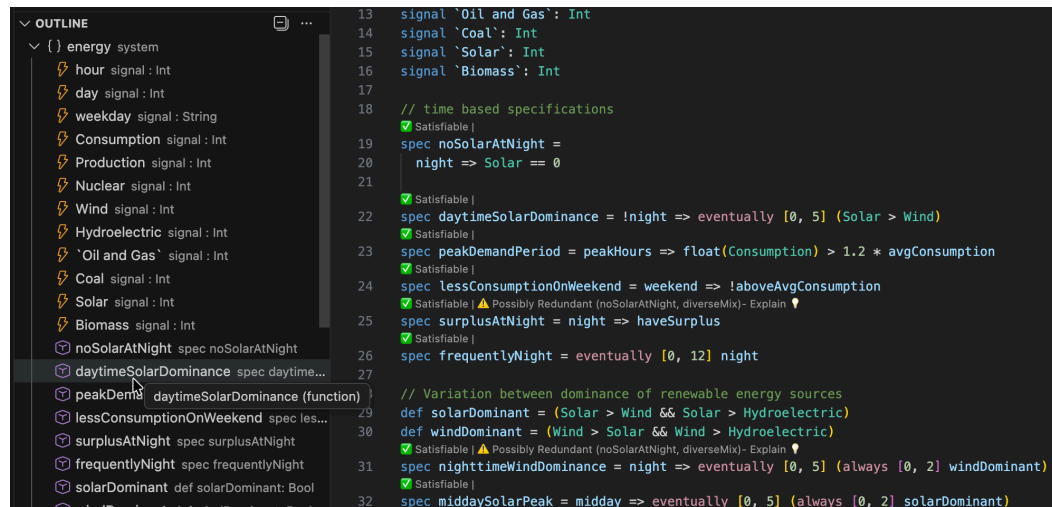
拡張機能は、仕様を記述する際にリアルタイムでチェックを実行します。エラーには下線が引かれます。該当箇所のコードにカーソルを合わせると、エラーが表示されます：



拡張機能は構文エラー、型エラーなどをチェックします。

## ドキュメントアウトライン

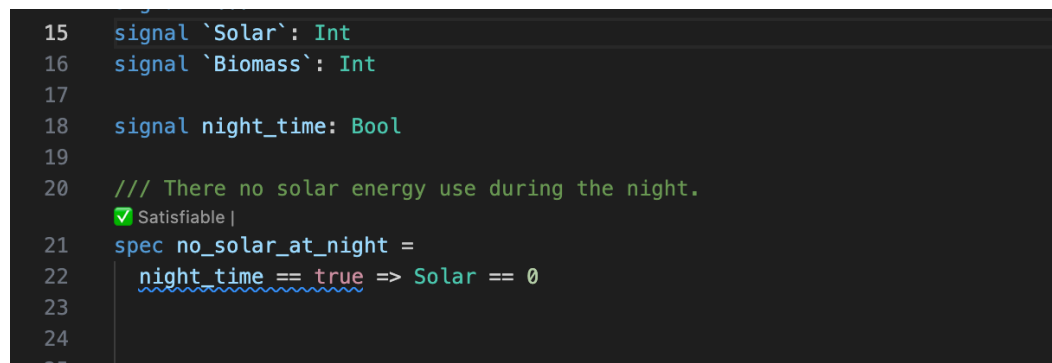
拡張機能は、仕様ファイルの階層的なアウトラインを提供します：



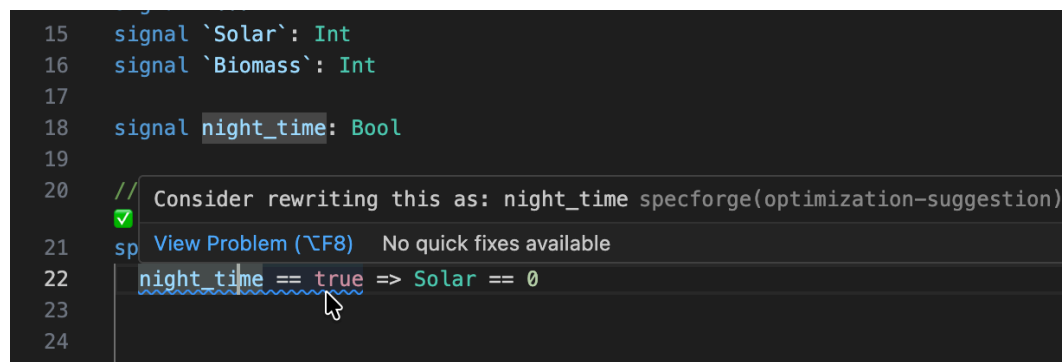
- VSCodeのエクスプローラーサイドバーで「アウトライン」ビューを開く
- すべての仕様、定義、シグナル、およびパラメータを一目で確認
- 任意のシンボルをクリックして定義にジャンプ
- アウトラインは自動的に更新されます

## 診断

拡張機能は自動的にさまざまなチェックを実行し、フィードバックを提供します。



診断にカーソルを合わせると、メッセージが表示されます。



診断には以下が含まれます：

- 未使用の `signal`、`param`、`def` などの警告
- 最適化の提案
- 間隔で時間依存の式を使用することに関する警告（設定されている場合）

## コードレンズ

コードレンズは、コード内の仕様の上に表示される対話型ボタンで、警告などの情報と（場合により）それに応じてできることを提示します。

### 充足可能性チェック

各仕様の上に、仕様が充足可能かどうかを示すコードレンズが表示されます：

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

以下はSpecForgeが充足不可能である可能性があると検出した仕様の例です：

```
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))
```

ユーザーは説明を求めることができます：

```
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))
```

ⓘ solarPeak requires a future moment where Solar strictly dominates Wind and Hydroelectric and, at the same time, Wind strictly dominates Solar and Hydroelectric; these two conditions cannot both hold, so the inner predicate is always false and the spec is unsatisfiable.

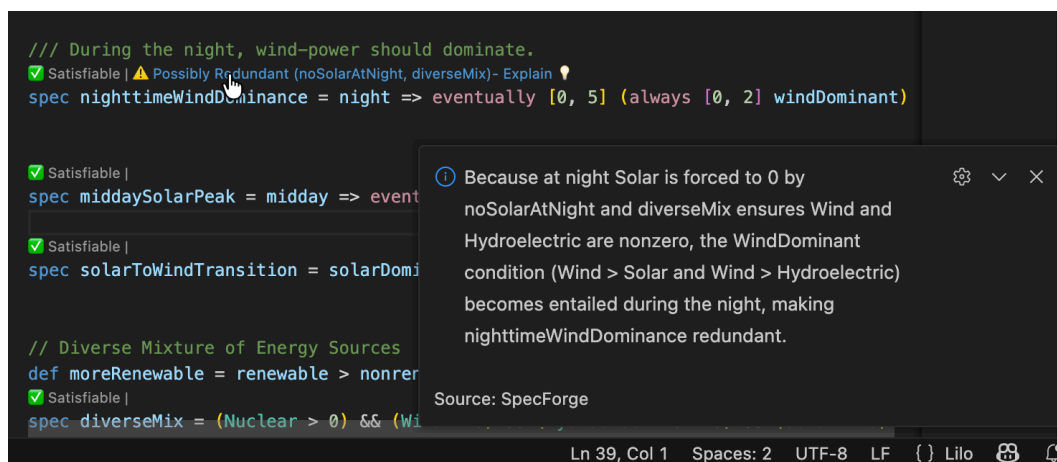
SpecForgeが充足可能性を判断できなかった場合、タイムアウトを長くして解析を再試行することができます。

### 冗長性

システムが仕様が冗長である可能性があると検出した場合、警告がコードレンズとして表示されます：

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

この仕様は `noSolarAtNight` と `diverseMix` によって暗黙的に必ず成り立つため、書く必要がないということをSpecForgeが示しています。 `Explain` をクリックすると、冗長性の説明が生成されます：



The screenshot shows a VS Code editor with a code snippet. A tooltip is displayed over the code, providing an explanation for a redundancy. The code snippet includes comments and specifications for energy system components. The tooltip text explains that at night, solar is forced to 0 by `noSolarAtNight` and `diverseMix`, ensuring wind and hydroelectric are non-zero, making the `nighttimeWindDominance` condition redundant.

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

✓ Satisfiable |
spec middaySolarPeak = midday => event

✓ Satisfiable |
spec solarToWindTransition = solarDom

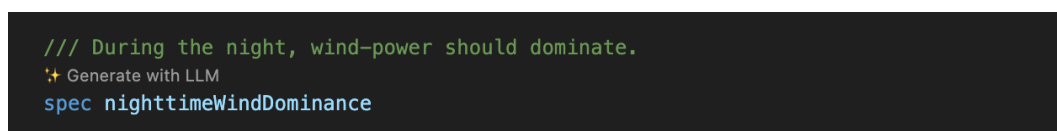
// Diverse Mixture of Energy Sources
def moreRenewable = renewable > nonrenewable
✓ Satisfiable |
spec diverseMix = (Nuclear > 0) && (Wind > 0) && (Hydroelectric > 0)
```

Because at night Solar is forced to 0 by `noSolarAtNight` and `diverseMix` ensures Wind and Hydroelectric are non-zero, the WindDominant condition (Wind > Solar and Wind > Hydroelectric) becomes entailed during the night, making `nighttimeWindDominance` redundant.

Source: SpecForge

## 仕様スタブ

`spec` が本体を持たない場合、それは仕様スタブです。この場合、コードレンズはAIを使用した仕様生成を提案します。




The screenshot shows a VS Code editor with a code snippet. A button labeled 'Generate with LLM' is visible next to the code, indicating a suggestion for generating the full specification from a stub.

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

`Generate with LLM` をクリックすると、現在のシステムで機能する仕様の定義が生成されます。仕様があまりにも曖昧である場合、または生成に他の障害がある場合は、エラーメッセージが表示されます。

## 仕様ステータスペイン

仕様ステータスパネルにアクセスするには、VSCodeの左側にあるSpecForgeアイコンをクリックします：



The screenshot shows the VS Code interface with the SpecForge sidebar open on the left. The sidebar lists project files and an outline of the energy system. The main editor shows a code snippet with signal definitions and specifications.

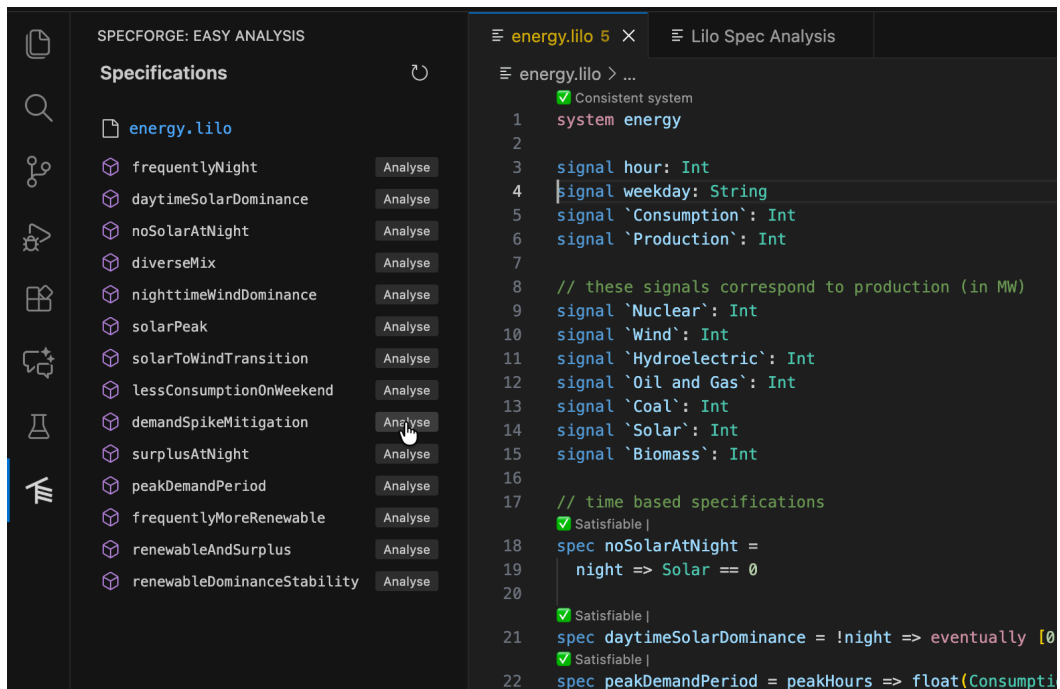
**SpecForge Sidebar:**

- pyproject.toml
- romania.csv
- sampled.csv
- OUTLINE
  - { } energy system
    - hour signal : Int
    - SpecForge signal : Int
    - weekday signal : String
    - Consumption signal : Int
    - Production signal : Int
    - Nuclear signal : Int
    - Wind signal : Int
    - Hydroelectric signal : Int

**Code Snippet:**

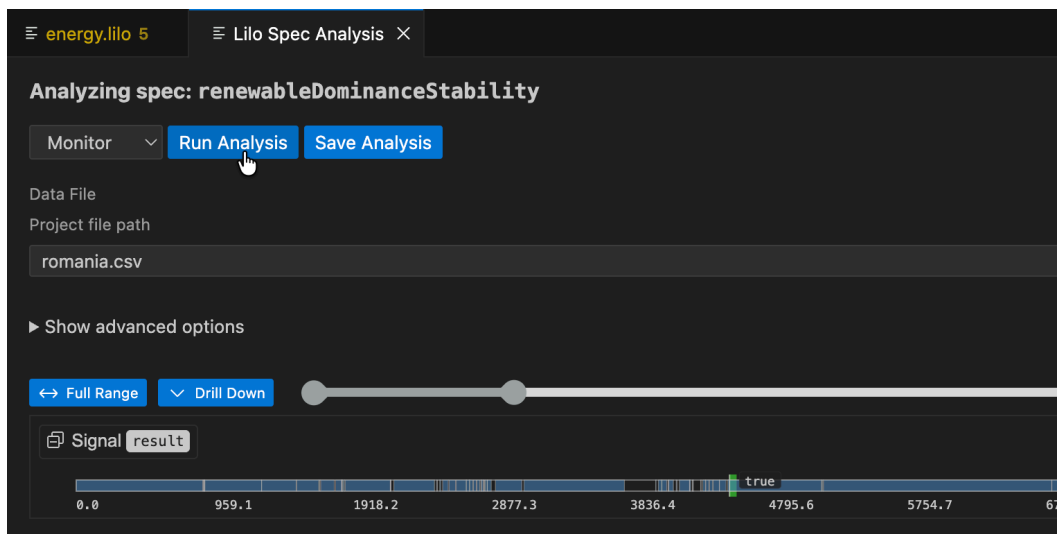
```
7 signal Production : Int
8
9 // these signals correspond to
10 signal `Nuclear`: Int
11 signal `Wind`: Int
12 signal `Hydroelectric`: Int
13 signal `Oil and Gas`: Int
14 signal `Coal`: Int
15 signal `Solar`: Int
16 signal `Biomass`: Int
17
18 // time based specifications
19 ✓ Satisfiable | ⚠ Possibly Redundant (solarPe
20 spec noSolarAtNight =
21   night => Solar == 0
22
23 ✓ Satisfiable | ⚠ Possibly Redundant (solarPe
24 spec daytimeSolarDominance = !n
```

サイドバーには、すべての仕様ファイルと定義されている仕様がリストされます：

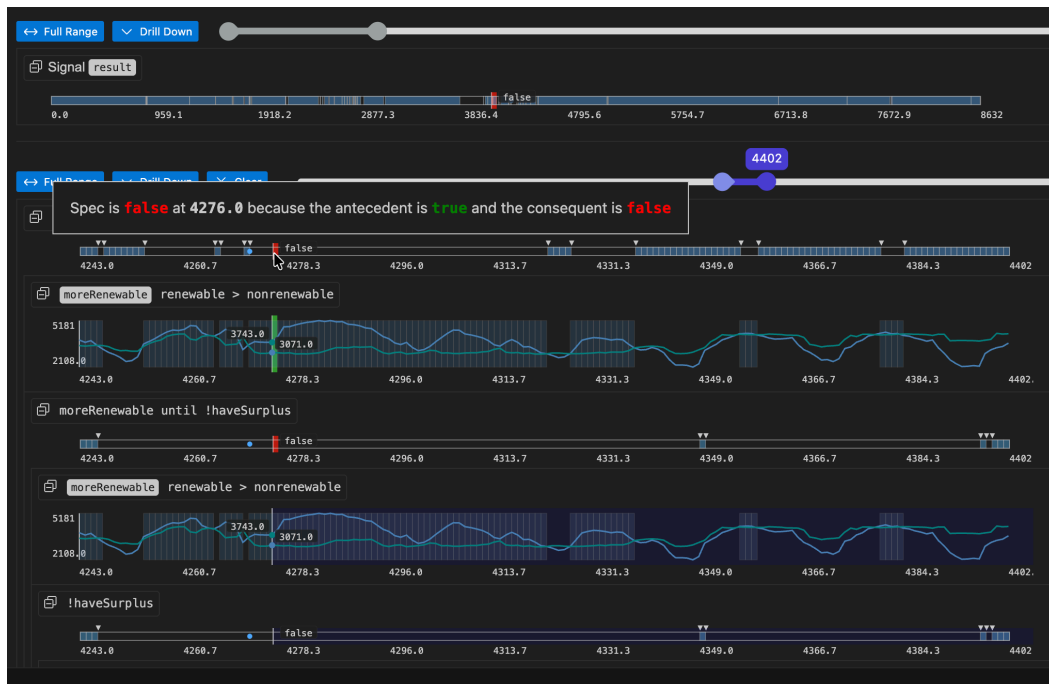


仕様の横にある Analyze をクリックすると、仕様解析ウィンドウが表示されます。これを使用して、さまざまな仕様解析タスクを起動できます：モニタリング、実証化、エクスポート、アニメーション、および反証化。

たとえば、仕様をモニタリングするには、ドロップダウンから Monitor を選択し、モニタリングするデータファイルを選択して、Run Analysis をクリックします。

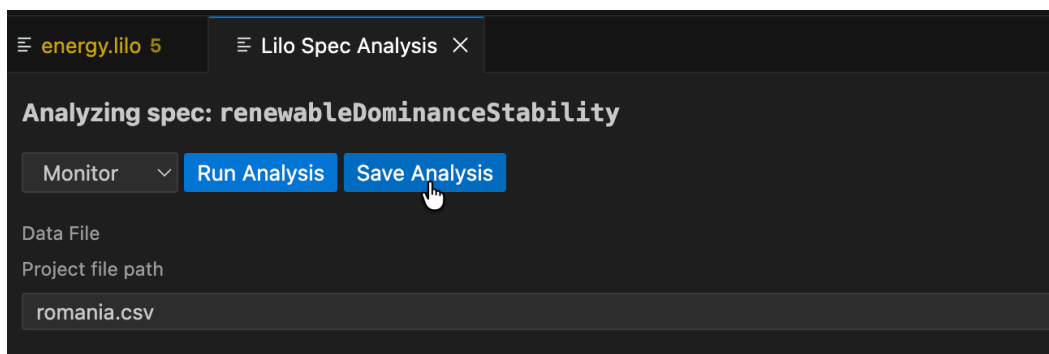


解析の結果が表示されます。青い領域は、仕様が真である時間を表します。大きなデータファイルの場合、ブール値の結果のみが表示されます。仕様がある時点で偽である理由をよりよく理解するには、ポイントを選択して Drill Down をクリックします。



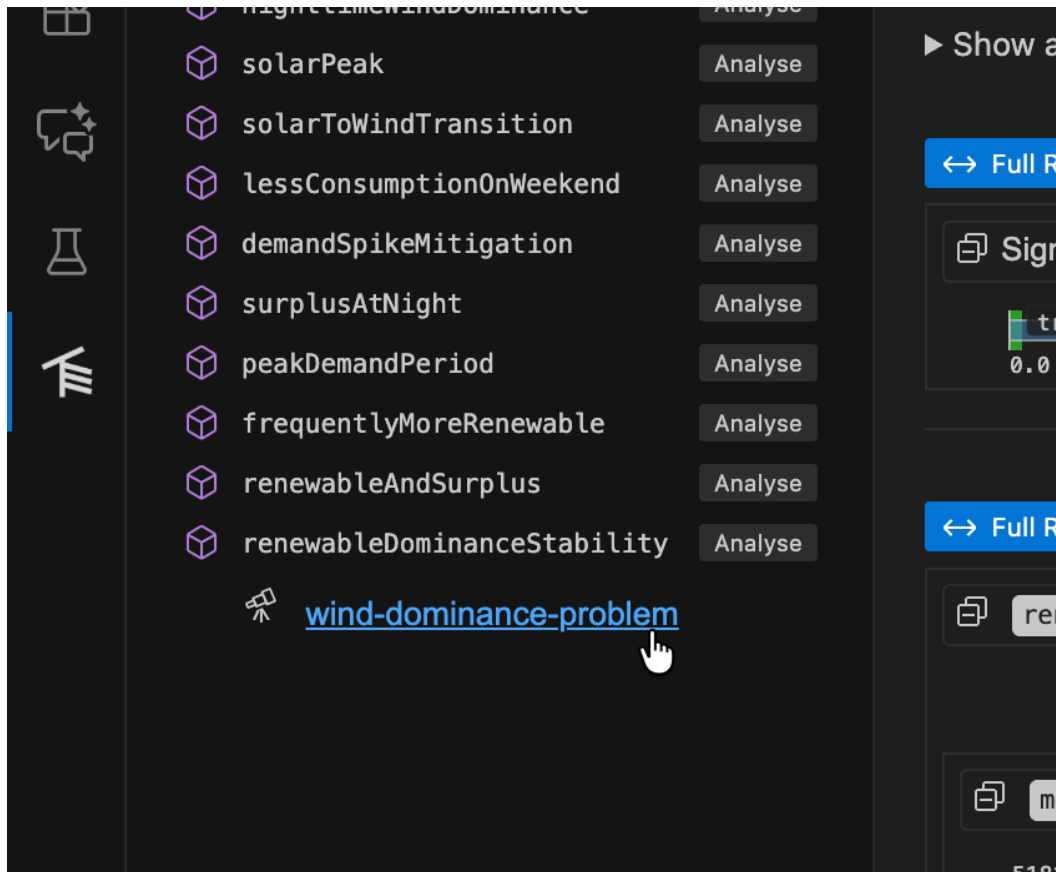
ドリルダウンは、デバッグツリーを表示するために、完全なデータソースの十分に小さいセグメントを選択しました。これは、仕様全体のモニタリング結果のツリーを表示します。各ノードは折りたたみまたは展開できます。タイムライン上にカーソルを合わせると、サブ式の結果タイムラインで関連する領域も強調表示されます。タイムライン上にカーソルを合わせると、そのサブ式のその時点での結果が真または偽である理由の説明が表示されます。

このような仕様解析は、Save Analysis ボタンをクリックすることで保存できます。



プロジェクト内で解析を保存する場所を選択します。

保存された解析は、関連する仕様の下での仕様ステータスサイドパネルに表示されます。



## 解析タイプ

GUIは5種類の解析をサポートしています：

### 1. Monitor

記録されたシステムの動作が仕様を満たしているかどうかをチェックします。

入力：

- **Signal Data**：時系列データを含むCSV、JSON、またはJSONLファイル
  - CSV：列ヘッダーはシグナル名と一致する必要があります
  - JSON/JSONL：シグナル名と一致するキーを持つオブジェクト
- **Parameters**：パラメータ値を含むJSONオブジェクト
- **Options**：モニタリング設定（[モニタリングオプション](#)を参照）

出力：

- 時間経過に伴う仕様の充足を示すモニタリングツリー
- サブ式へのドリルダウン
- シグナル値の視覚化

例：

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

### 2. Exemplify

仕様を満たす例のトレースを生成します。

入力：

- **Number of Points**：生成する時間ポイントの数（デフォルト：10）
- **Timeout**：生成に費やす最大時間（デフォルト：5秒）
- **Also Monitor**：生成されたトレースのモニタリングツリーも表示するかどうか
- **Assumptions**：満たすべき追加の制約（仕様式の配列）

出力：

- 時系列として生成されたシグナルデータ
- 「Also Monitor」が有効な場合のオプションのモニタリングツリー
- 生成されたデータのCSV/JSONエクスポート

ユースケース：

- 有効な動作がどのようなものかを理解する
- リアルなデータで他のコンポーネントをテストする
- テストフィクスチャの作成
- 仕様が意味をなすことを検証する

### 3. Falsify

外部モデルを使用して、仕様に違反する反例を検索します。

前提条件：

- `lilo.toml` に反証スクリプトを登録する必要があります：

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

反証スクリプトプロトコル：

スクリプトは次のコマンドライン引数を受け取ります：

- `--system`：システム名
- `--spec`：仕様名
- `--options`：オプションを含むJSON文字列
- `--params`：パラメータ値を含むJSON文字列
- `--project-dir`：プロジェクトルートへのパス

スクリプトは以下を行う必要があります：

1. 仕様に従ってシステムをシミュレートする
2. 仕様に違反するトレースを検索する
3. 成功または失敗のいずれかで正しい形式のJSONを出力する

入力：

- **Falsifier**：設定された反証器から選択（ドロップダウン）
- **Timeout**：反証の最大時間（デフォルト：240秒）
- **Parameters**：パラメータ値を含むJSONオブジェクト

出力：

- 反例が見つかった場合：
  1. 失敗したトレースを示す反証結果
  2. 反例の自動モニタリング
  3. 仕様が失敗する場所/方法の視覚化
- 反例が見つからなかった場合：成功メッセージ

スクリプトが実行可能であることを確認してください：



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

## 4. Export

仕様を他の形式に変換します。

エクスポート形式：

- **Lilo**：オプションの変換を伴う `.lilo` 形式としてエクスポート
- **JSON**：機械可読のJSON表現

入力：

- **Export Type**：ターゲット形式を選択
- **Parameters**：パラメータ値（エクスポートに必要な場合）

出力：

- 選択された形式でエクスポートされた仕様
- ファイルに保存できます

ユースケース：

- 他のツールとの統合
- ドキュメント生成
- 仕様のアーカイブ

## 5. Animate

時間の経過とともに仕様の動作を示すアニメーションを作成します。

入力：

- **SVG Template**：プレースホルダーを含むSVGファイルへのパス
- **Signal Data**：時系列データを含むCSV、JSON、またはJSONLファイル

出力：

- システムの進化を示すフレームごとのSVG画像
- アニメーション化された視覚化に組み合わせることができます

SVGテンプレート形式：

SVGテンプレートには、シグナル値の `data-` 属性を含める必要があります。例：

```
<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>
```

この例では、`<circle>` 要素の `cy` 属性は、`data-cy="temperature"` 属性のおかげで、`temperature` シグナルの値によってアニメーション化されます。

## モニタリングオプション

MonitorまたはFalsify解析を実行する際に、次のオプションを構成できます：

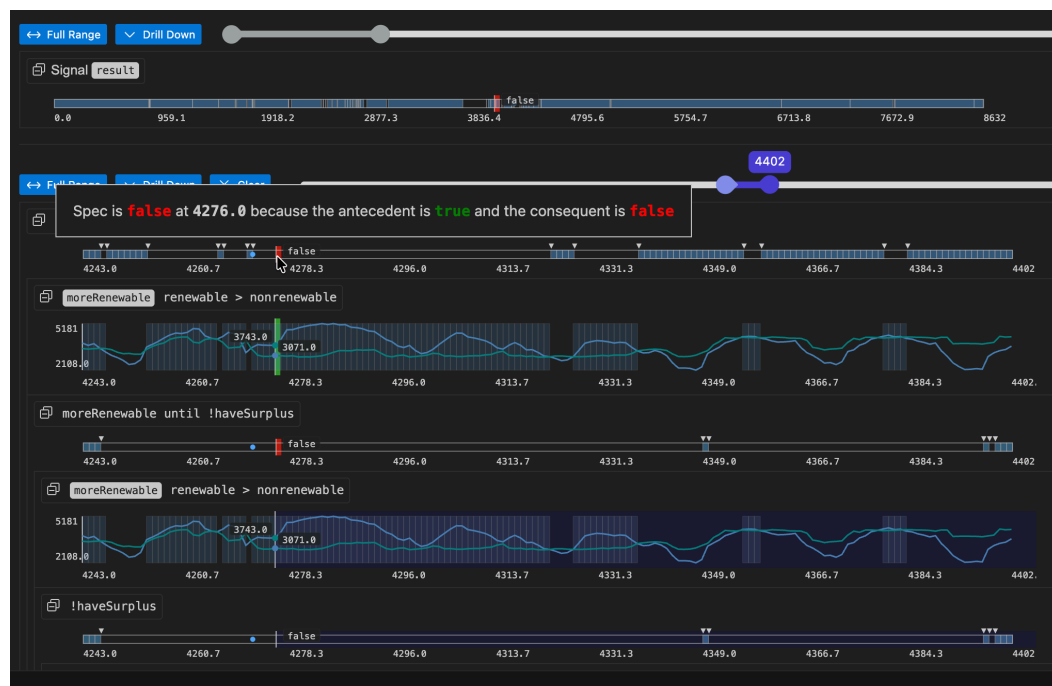
- **Time Bounds**：モニタリングを特定の時間範囲に制限
- **Sampling**：時間解像度を調整
- **Signal Filtering**：特定のシグナルのみをモニタリング
- （追加のオプションが利用可能な場合があります）

## 結果の操作

### モニタリングツリー

モニタリングツリーには以下が表示されます：

- **Root**：全体的な仕様の結果（真/偽/不明）
- **Sub-expressions**：仕様が真または偽である理由をドリルダウン
- **Timeline**：任意の式にカーソルを合わせて、いつ真/偽であるかを確認
- **Highlighting**：カーソルを合わせると、関連するセグメントが強調表示されます



### 保存された解析の読み込み

保存された `.analysis.sf` ファイルを開くと、次のことができます：

- 元の設定を確認する
- 同じ設定で解析を再実行する
- パラメータを変更して再実行する
- 結果をエクスポートする

解析エディターは、メインの解析GUIと同じインターフェースを提供しますが、保存された設定で事前に入力されています。

解析はファイルです。解析を変更した場合は、保存する必要があります。

## Jupyterノートブック統合

拡張機能には、JupyterノートブックでSpecForgeの結果を表示するためのノートブックレンダラーが含まれています。

## アクティベーション

レンダラーは次の場合に自動的にアクティブになります：

- Jupyter ノートブック（.ipynb ファイル）
- VSCode 対話型 Python ウィンドウ

## Python SDKでの使用

SpecForge Python SDKを使用する場合、結果は自動的にレンダリングされます：

```
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はノートブックで自動的にレンダリングされます
result
```

## スニペット

拡張機能は、一般的なSpecForge操作（`monitor`、`exemplify`、`export`）のためのPythonコードスニペットを提供します。スニペット名を入力してTabキーを押すと、プレースホルダーを挿入してナビゲートできます。

## トラブルシューティング

### 拡張機能が動作しない

SpecForgeサーバーを確認してください：

- サーバーが実行中であることを確認します（[SpecForgeのセットアップ](#)を参照）
- 設定のAPI base URLがサーバーと一致することを確認します
- サーバーログでエラーを確認します

言語サーバーを再起動してください：

- コマンドパレットから `SpecForge: Restart Language Server` を実行します
- 「出力」パネル（表示 → 出力）を確認し、ドロップダウンから「SpecForge」を選択します

### 診断が表示されない

更新をトリガーしてください：

- ファイルを保存します（Ctrl+S / Cmd+S）
- ファイルを閉じて再度開きます
- 言語サーバーを再起動します

サーバー接続を確認してください：

- ステータスバーで接続ステータスを確認します
- 設定されたURLでサーバーにアクセスできることを確認します

## コードレンズが表示されない

設定を確認してください：

- VSCodeでコードレンズが有効になっていることを確認します： `Editor > Code Lens`
- ファイルを保存してコードレンズの計算をトリガーします

エラーを確認してください：

- 解析を妨げる解析エラーまたは型エラーを探します
- コード内の赤い波線を修正します

## 解析GUIが読み込まれない

webviewを確認してください：

- 開発者ツールを開きます： ヘルプ > 開発者ツールの切り替え
- コンソールタブでエラーを探します
- webview iframeが読み込まれたかどうかを確認します

サーバー接続を確認してください：

- API base URLが正しいことを確認します
- ブラウザでURLをテストします (JSON応答が表示されるはずです)

## 反証スクリプトエラー

スクリプトのセットアップを確認してください：

- `lilo.toml` のスクリプトパスを確認します
- スクリプトが実行可能であることを確認します： `chmod +x script.py`
- 例の引数でスクリプトを手動でテストします

スクリプト出力を確認してください：

- スクリプトは有効なJSONを出力する必要があります
- 正しい結果形式を使用します ([Falsify](#)を参照)
- エラーメッセージのためにスクリプトログを確認します

一般的な問題：

- `ENOENT`：スクリプトファイルが見つかりません (パスを確認)
- `EACCES`：スクリプトが実行可能ではありません (`chmod +x` を実行)
- 解析エラー：無効なJSON出力 (スクリプト出力形式を確認)

# SpecForge Python SDK

SpecForge Python SDKは、SpecForge APIと連携し、形式仕様のモニタリング、アニメーション、エクスポート、および例示を可能にするためのSDKです。

Python環境でのSDKのインストールと設定については、[Python SDKのセットアップガイド](#)を参照してください。

## クイックスタート

```
from specforge_sdk import SpecForgeClient

# クライアントの初期化
specforge = SpecForgeClient(base_url="http://localhost:8080")

# APIのヘルスチェック（接続確認）
if specforge.health_check():
    print("✓ SpecForge APIに接続しました")
    print(f"APIバージョン: {specforge.version()}")
else:
    print("✗ SpecForge APIに接続できません")
```

## 主な機能

SDKは、SpecForgeの主要な機能へのアクセスを提供します：

- **モニタリング**: データに対して仕様をチェック
- **アニメーション**: 時系列の可視化を作成
- **エクスポート**: 仕様を異なる形式に変換
- **例示**: 仕様を満たすサンプルデータを生成

## ドキュメント

包括的なデモノートブック `sample-project/demo.ipynb` をご覧ください。以下の内容が含まれています：

- 詳細な使用例
- Jupyter notebookとの統合
- カスタムレンダリング機能

## APIメソッド

### `SpecForgeClient(base_url, ...)`

SpecForge APIクライアントを初期化します。

パラメータ:

- `base_url` (str): SpecForge APIサーバーのベースURL（デフォルト: "http://localhost:8080"）
- `project_dir` (str/Path): オプションのプロジェクトディレクトリパス。指定しない場合、カレントディレクトリから `lilo.toml` を検索します
- `timeout` (int): リクエストのタイムアウト秒数（デフォルト: 30）

- `check_version` (bool): 初期化時にバージョンの不一致をチェックするかどうか（デフォルト: True)

例:

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

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

データに対して仕様をモニタリングします。評価結果とオプションでロバストネスメトリクスを返します。

主なパラメータ:

- `system` (str): 定義を含むシステム
- `definition` (str): モニタリングする仕様の名前
- `data_file` (str/Path): データファイルのパス (CSV、JSON、またはJSONL)
- `data` (list/DataFrame): 辞書のリストまたはpandas DataFrameとしての直接データ
  - **注意:** `data_file` または `data` のいずれか1つのみを指定してください
- `params_file` (str/Path): システムパラメータファイルのパス
- `params` (dict): 辞書としての直接システムパラメータ
  - **注意:** `params_file` または `params` のうち最大1つを指定してください
- `encoding` (dict): レコードエンコーディング設定
- `verdicts` (bool): 評価情報を含める (デフォルト: True)
- `robustness` (bool): ロバストネス解析を含める (デフォルト: False)
- `return_timeseries` (bool): Trueの場合DataFrameを返す、Falseの場合JSONを表示 (デフォルト: False)

例:

```
# データファイルとパラメータファイルでモニタリング
specforge.monitor(
    system="temperature_sensor",
    definition="always_in_bounds",
    data_file="sensor_data.csv",
    params_file="temperature_config.json"
)

# データファイルとパラメータ辞書でモニタリング
specforge.monitor(
    system="temperature_sensor",
    definition="always_in_bounds",
    data_file="sensor_data.csv",
    params={"min_temperature": 10.0, "max_temperature": 24.0}
)

# DataFrameでモニタリングし、結果をDataFrameとして取得
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
)

# ロバストネス解析付きでモニタリング
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, ...)**

仕様、データ、SVGテンプレートからアニメーションを作成します。

### **主なパラメータ:**

- `system` (str): アニメーション定義を含むシステム
- `svg_file` (str/Path): SVGテンプレートファイルのパス
- `data_file` (str/Path): データファイルのパス
- `data` (list/DataFrame): 辞書のリストまたはpandas DataFrameとしての直接データ
  - **注意:** `data_file` または `data` のいずれか1つのみを指定してください
- `encoding` (dict): レコードエンコーディング設定
- `return_gif` (bool): Trueの場合、base64エンコードされたGIF文字列を返す (デフォルト: False)
- `save_gif` (str/Path): GIFファイルを保存するオプションのパス

### **例:**

```
# Jupyterでアニメーションフレームを表示
specforge.animate(
    system="temperature_sensor",
    svg_file="temp.svg",
    data_file="sensor_data.csv"
)

# アニメーションをGIFファイルとして保存
specforge.animate(
    system="scene",
    svg_file="scene.svg",
    data_file="scene.json",
    save_gif="output.gif"
)

# GIFデータをbase64文字列として取得
gif_data = specforge.animate(
    system="temperature_sensor",
    svg_file="temp.svg",
    data=synthetic_df,
    encoding=nested_encoding(),
    return_gif=True
)
```

## **export(system, definition, ...)**

仕様を異なるフォーマット (例: LILO形式) にエクスポートします。

### **主なパラメータ:**

- `system` (str): 定義を含むシステム
- `definition` (str): エクスポートする仕様の名前
- `export_type` (dict): エクスポート形式の設定 (デフォルト: LILO)
- `params_file` (str/Path): システムパラメータファイルのパス
- `params` (dict): 辞書形式で直接指定されたシステムパラメータ
  - **注意:** `params_file` または `params` のうち最大1つを指定してください
- `encoding` (dict): レコードエンコーディング設定
- `return_string` (bool): Trueの場合、エクスポートされた文字列を返す。Falseの場合JSONを表示 (デフォルト: False)

### **例:**

```

# LILO形式に文字列としてエクスポート
lilo_result = specforge.export(
    system="temperature_sensor",
    definition="always_in_bounds",
    export_type=EXPORT_LILO,
    return_string=True
)
print(lilo_result)

# パラメータファイルでエクスポート
export_result = specforge.export(
    system="temperature_sensor",
    definition="always_in_bounds",
    export_type=EXPORT_LILO,
    params_file="temperature_config.json",
    return_string=True
)

# パラメータ辞書でエクスポート
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
)

# JSON形式にエクスポート (Jupyterで表示)
specforge.export(
    system="temperature_sensor",
    definition="humidity_correlation",
    export_type=EXPORT_JSON
)

```

## exemplify(system, definition, ...)

仕様を満たすサンプルデータを生成します。

主なパラメータ:

- system (str): 定義を含むシステム
- definition (str): 例示する仕様の名前
- assumptions (list): 生成を制約する追加の仮定 (デフォルト: [])
- n\_points (int): 生成するデータポイント数 (デフォルト: 10)
- params\_file (str/Path): システムパラメータファイルのパス
- params (dict): 辞書としての直接システムパラメータ
  - **注意:** params\_file または params のうち最大1つを指定してください
- params\_encoding (dict): パラメータのレコードエンコーディング
- timeout (int): 例示のタイムアウト秒数 (デフォルト: 30)
- also\_monitor (bool): 生成されたデータもモニタリングするかどうか (デフォルト: False)
- return\_timeseries (bool): Trueの場合、DataFrameを返す。Falseの場合、JSONを表示 (デフォルト: False)

例:



```

# 20サンプルの例を生成
specforge.exemplify(
    system="temperature_sensor",
    definition="always_in_bounds",
    n_points=20
)

# 仮定付きで例を生成
humidity_assumption = {
    "expression": "eventually (humidity > 25)",
    "rigidity": "Hard"
}
specforge.exemplify(
    system="temperature_sensor",
    definition="humidity_correlation",
    assumptions=[humidity_assumption],
    n_points=20
)

# 部分的なパラメータで例を生成（ソルバーが不足値を埋める）
specforge.exemplify(
    system="temperature_sensor",
    definition="humidity_correlation",
    assumptions=[{"expression": "always_in_bounds", "rigidity": "Hard"}],
    params={"min_temperature": 38.0},
    n_points=20
)

# 例を生成してDataFrameとして取得
example_df = specforge.exemplify(
    system="temperature_sensor",
    definition="temperature_in_bounds",
    n_points=15,
    also_monitor=False,
    return_timeseries=True
)

```

## health\_check()

SpecForge APIが利用可能で応答しているかどうかをチェックします。

**戻り値:** bool - APIが正常な場合True、それ以外の場合False

**例:**

```

if specforge.health_check():
    print("✓ SpecForge APIに接続しました")
else:
    print("✗ SpecForge APIに接続できません")

```

## version()

APIサーバーのバージョンとSDKバージョンを取得します。

**戻り値:** "api" と "sdk" のキーを持つ dict

**例:**

```

versions = specforge.version()
print(f"APIバージョン: {versions['api']}")
print(f"SDKバージョン: {versions['sdk']}")

```

## 対応ファイル形式

- **仕様:** .lilo ファイル

- データ: .csv, .json, .jsonl ファイル
- 可視化: .svg ファイル

## 必要要件

- Python 3.12+
- requests>=2.25.0
- urllib3>=1.26.0

# SpecForge SDK サンプルプロジェクト

このディレクトリには、SpecForge Python SDKの機能を実演する完全な例が含まれています。

## ファイル

- `temperature_monitoring.lilo` - 温度監視のためのサンプル仕様
- `sensor_data.csv` - サンプルセンサーデータ（温度と湿度を含む31のデータポイント）
- `demo.ipynb` - すべてのSDK機能を実演するJupyterノートブック
- `temperature_config.json` - 温度監視例のための設定ファイル（パラメーター）
- `util.lilo` - サンプル仕様のためのユーティリティ関数

## セットアップ

**Note:** 以下、ユーザーは `temperature_sensor` フォルダをVSCodeで開くことを推奨します。

### 1. 仮想環境の作成とアクティベート

一般的に（必須ではありませんが）、仮想環境でこれを行うことが推奨されます。以下の手順に従ってください：

```
# サンプルプロジェクトディレクトリに移動
cd temperature_sensor

# 仮想環境を作成
python -m venv .venv

# 仮想環境をアクティベート
# Windowsの場合：
.venv\Scripts\activate
# macOS/Linuxの場合：
source .venv/bin/activate
```

### 2. 依存関係のインストール

```
# SpecForge SDK Wheelをインストール
pip install ../specforge_sdk-xxx.whl

# サンプルプロジェクトの追加依存関係をインストール
pip install jupyter pandas matplotlib numpy
```

### 3. インストールの確認

```
# SDKが正しくインストールされたかを確認
python -c "from specforge_sdk import SpecForgeClient; print('✓ SDKが正常にインストールされました')"
```

## サンプル例の実行

### 前提条件

1. SpecForge APIサーバーが `http://localhost:8080/health` で実行中であることを確認してください。
2. 仮想環境をアクティベートしてください\*\*（まだアクティブでない場合）：

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

### サンプル例の実行

拡張機能が機能していることを確認するには、`demo.ipynb` のセルを実行します。監視コマンドの出力には視覚化が含まれているはずです。

ノートブックが正しいPythonカーネルに接続されていることを確認する必要があるかもしれません。多くの場合、`.venv` (Python3.X.X) または `ipykernel` です。これはVSCodeのノートブックインターフェースから設定できます。

## サンプルデータの概要

含まれている `sensor_data.csv` には以下が含まれています：

- 31の時点（0.0から30.0）
- 温度の読み取り値（20.8°Cから25.0°C）
- 湿度の読み取り値（40.9%から51.5%）

このデータは、温度の境界、安定性、および湿度との相関を含むさまざまな仕様をテストするために設計されています。

## 環境の非アクティベート

サンプルプロジェクトでの作業が完了したら：

```
deactivate
```

## トラブルシューティング

### よくある問題

1. **「Module not found」エラー**: 仮想環境がアクティベートされ、SDKが `pip install ../specforge_sdk-xxx.whl` でインストールされていることを確認してください。
2. **接続拒否**: SpecForge APIサーバーが `http://localhost:8080/health` で実行されていることを確認してください。
3. **Jupyterが見つからない**: アクティベートされた仮想環境で `pip install jupyter` を実行してインストールしてください。
4. **サンプルファイルが見つからない**: `temperature_sensor` ディレクトリにいることを確認してください。

## セットアップの確認

```
# Python環境を確認
```

```
which python
```

```
pip list | grep specforge
```

```
# API接続をテスト
```

```
python -c "from specforge_sdk import SpecForgeClient; client = SpecForgeClient();
```

```
print('Health check:', client.health_check())"
```

# 変更履歴

すべての注目すべき変更はここに文書化されます。フォーマットは[Keep a Changelog](#)に基づいています。Liloは[セマンティックバージョンング](#)に準拠しています。

## v0.5.4 - 2025-11-20

### 追加

- ローカルシグナルファイル監視
- Docker向けオフラインライセンス
- 監視のドリルダウン
- 監視の注目ポイント
- [VSCode拡張機能](#)ドキュメント
- GHCRで利用可能なパブリックDockerイメージと付属ドキュメント
- 解析サイドバーが自動更新され、解析実行中にスピナーを表示するようになりました

### 変更

- 監視ツリーのサンプル上限が3100に引き上げられました
- 仕様ステータスがプレビュー機能ではなくなりました

### 修正

- 監視ツリーの範囲のレスポンスシブ化
- VSCodeサイドバーのスタイル
- 監視用サンプルプロジェクトが正しくバンドルされるようになりました
- ローカルシグナルファイルのUX問題を解決

## v0.5.3 - 2025-11-14

### 追加

- [Whirlwindツアーガイド](#)
- SpecForgeのオフラインライセンス
- 監視の自動ダウンサンプリング
- GeminiおよびOllama LLMプロバイダーサポート
- CLI監視コマンドに間隔とサンプリングオプションを追加
- 反証例をドキュメントに追加

### 修正

- 古い反証器リストの問題を解消
- モジュール名不一致のエラーに関する報告場所を修正
- CLIコマンドがプロジェクトディレクトリから実行されるようになりました

## v0.5.2 - 2025-11-10

### 変更

- 反証タイムアウトを60秒から240秒に変更

### 修正

- 複数ファイルのエラー報告

## v0.5.1 - 2025-11-05

### 追加

- 新しいドキュメントサイト：<https://docs.imiron.io/>。
- gifアニメーションを作成できるようになりました。
- プロジェクトは `lilo.toml` ファイルで設定できるようになりました。[プロジェクト設定](#)を参照してください。
- システム反証器 (system falsifier) を `lilo.toml` で登録できるようになりました。
- VSCodeの仕様ステータスに解析結果が表示されるようになりました。
- VSCodeでの仕様解析。
- 仕様解析ペインから反証エンジン (falsification engine) を実行できるようになりました。

### 変更

- レコード構築/更新の競合に対するより良い型エラー。
- LLMによる説明はユーザーのVSCode設定に従ってローカライズされるようになりました。
- 未定義変数のエラーで、該当スコープにある似た名前の変数が列挙されるようになりました。
- システムのグローバル変数（`signal` と `param`）に、docstringを含め、属性を記述できるようになりました。
- コマンドJSON形式が大幅に変更され、今後は安定している（後方互換性がある）ことが期待されます。特に、ファイル名ではなくシステム名を使用します。
- パラメータに `null` (JSON) を指定してデフォルトのパラメータ値を ~~削除~~ できるようになりました。
- LLM仕様生成は、不十分な指定の仕様に対して失敗します。
- CLIインターフェースがモジュールで動作するように更新されました。

## v0.5.0 - 2025-10-14

### 追加

- デフォルト `param` : `param foo: Float = 42` はパラメータ `foo` のデフォルト値として `42` を設定します。
- タイムアウト属性：

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

は、`spec foo` の解析タスクのタイムアウトを3秒に設定します。

- サーバー/クライアントバージョンの不一致に対する警告。
- 仕様スタブ：

`spec no_overheat`

は「仕様スタブ」（未実装の仕様）を作成します。AIを使用して、docstringを使用した実装を提案するコードアクションもあります。

- より長いタイムアウトでの解析の再試行：解析がタイムアウトした場合、より長いタイムアウトで再試行するコードアクションがあります。
- レコード機能：
  - レコード更新（ディープを含む）
  - フィールドのパン。
  - パス構築とパス更新。
- 未使用の `def`、`signal`、`param` に対する警告。
- VSCodeのコード階層。
- モジュール：ユーザーはモジュール（`def` と `type` 宣言のみを含む）を作成し、インポートできます。

## 変更

- VSCodeのコードレンズは一度に1つずつ解決されるため、より応答性の高いエクスペリエンスになります。