

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。いくつかのサンプルプロジェクトは[サンプルリポジトリ](#)から入手するか、[リリースページ](#)からダウンロードできます。

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サーバー実行ファイル**：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。
- **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 `specforge.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。 `specforge.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

サーバー環境変数

SpecForgeサーバーは起動時に以下の環境変数を読み込みます。 `specforge serve` の実行前に設定してください。

変数	型	デフォルト	説明
SPECFORGE_PORT	Integer	8080	サーバーがリッスンするポート。
SPECFORGE_CACHE_BOUND	Integer	64	例示キャッシュの最大エントリ数。キャッシュはSMTソルバー（Z3）の結果を保存し、冗長な計算を回避

変数	型	デフォルト	説明
			します。0 に設定するとキャッシュを無効化します。
SPECFORGE_SEMAPHORE_LIMIT	Integer	8	同時に実行できるSMTソルバー（Z3）インスタンスの最大数。リソース枯渇を防ぐために並列実行を制限します。0 に設定するとセマフォを無効化します（並列数制限なし）。

使用例：

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

LLM関連の環境変数（SPECFORGE_LLM_PROVIDER、SPECFORGE_LLM_MODEL、OPENAI_API_KEY など）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。

これらすべての設定は、[SpecForge VSCode拡張機能の設定](#)でも構成できます。

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. エディションとライセンスの設定

利用水準もライセンスも設定されていない場合、SpecForgeはデフォルトでCommunity Editionとして動作します。commercial水準では、有効なライセンスによってEnterprise Editionが有効にならない限り、プロジェクトサイズに制限のあるEssential Editionとして動作します。詳細および利用水準の変更手順については、[エディション](#)、[利用水準](#)、[ライセンス](#)を参照してください。

Enterprise Editionを利用するには、`license.json` ファイルを次のいずれかの場所に配置します：

SpecForgeは、これらの `license.json` のパスより前に、標準設定ディレクトリとカレントディレクトリの `*-license.json` も検索します。 `SPECFORGE_LICENSE_FILE` を設定した場合は、そのファイルのみを検索します。

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から取得できます。

他のプロバイダー（Gemini、Anthropic、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。

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。

6. Python SDKのインストール（オプション）

Python SDKを使用すると、PythonからプログラムでSpecForgeサーバーと連携できます。PythonノートブックにSpecForgeの解析を組み込んだり、Pandas DataFrameを使ってデータの入出力を直接行ったりすることができます。セットアップ手順については[Python SDKセットアップガイド](#)を参照してください。

参考情報

- [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](#) - `specforge.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

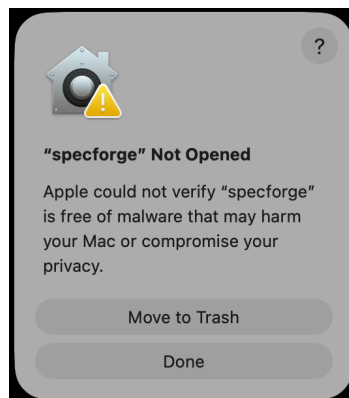
[SpecForgeリリース](#)ページから `specforge-x.y.z-macOS-ARM64.tar.bz2` をダウンロードし、任意のディレクトリに展開します。

ダウンロードしたSpecForgeバイナリの実行許可

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 --version
```

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。代わりにコマンドが正常に実行され、バージョンが表示された場合は、次のステップに進んでください。

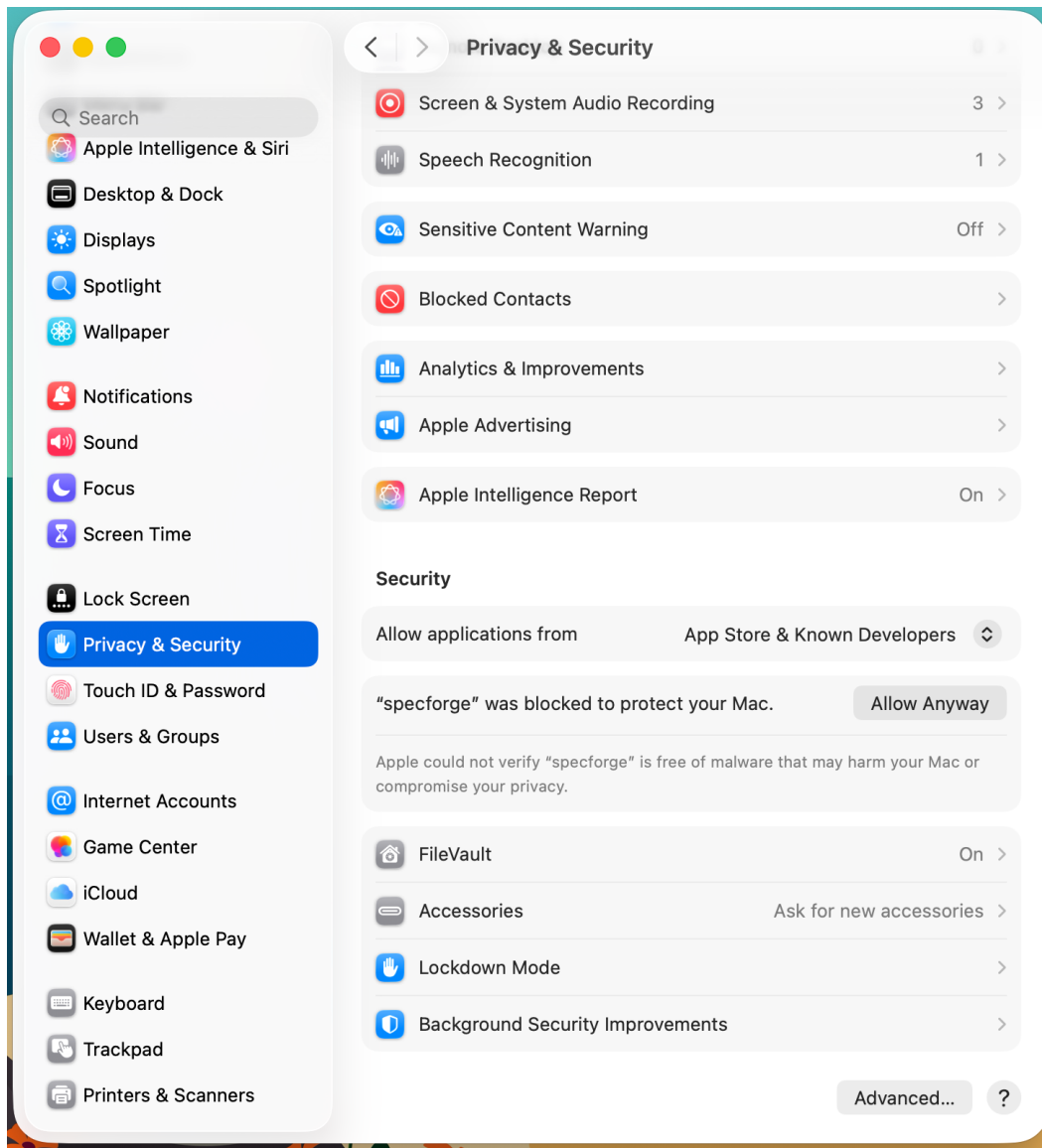


specforge実行ファイルをホワイトリストに追加するには、次のコマンドを実行します：

```
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。」と表示されるはずですが。
3. 「このまま開く」をクリックします。



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。**rsvg-converter**はオプションですが、アニメーション機能で使用するSVGからPNGへの変換のために推奨されます。

Homebrewを使用してインストールします：

```
brew install z3 librsvg
```

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

注意： librsvg は rsvg-convert コマンドを提供します。このパッケージがない場合、Python SDKの `animate()` 機能はSVGビジュアライゼーションからPNGフレームをレンダリングできません。その他のSpecForge機能はこのパッケージなしでも動作します。

3. エディションとライセンスの設定

利用水準もライセンスも設定されていない場合、SpecForgeはデフォルトでCommunity Editionとして動作します。commercial水準では、有効なライセンスによってEnterprise Editionが有効にならない限り、

プロジェクトサイズに制限のあるEssential Editionとして動作します。詳細および利用水準の変更手順については、[エディション](#)、[利用水準](#)、[ライセンス](#)を参照してください。

Enterprise Editionを利用するには、`license.json` ファイルを以下のいずれかの場所に配置します：

SpecForgeは、これらの `license.json` のパスより前に、標準設定ディレクトリとカレントディレクトリの `*-license.json` も検索します。 `SPECFORGE_LICENSE_FILE` を設定した場合は、そのファイルのみを検索します。

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から取得してください。

その他のプロバイダー（Gemini、Anthropic、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。

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。

7. Python SDKのインストール（オプション）

Python SDKを使用すると、PythonからプログラムでSpecForgeサーバーと連携できます。PythonノートブックにSpecForgeの解析を組み込んだり、Pandas DataFrameを使ってデータの入出力を直接行ったりすることができます。セットアップ手順については[Python SDKセットアップガイド](#)を参照してください。

参考情報

- [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](#) - `specforge.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。**rsvg-converter**はオプションですが、アニメーション機能で使用するSVGからPNGへの変換のために推奨されます。

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 librsvg2-bin
```

Fedora/RHEL：

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

Arch Linux：

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

注意： `librsvg2-bin` / `librsvg2-tools` / `librsvg` パッケージは `rsvg-convert` コマンドを提供します。このパッケージがない場合、Python SDKの `animate()` 機能はSVGビジュアライゼーションからPNGフレームをレンダリングできません。その他のSpecForge機能はこのパッケージなしでも動作します。

3. エディションとライセンスの設定

利用水準もライセンスも設定されていない場合、SpecForgeはデフォルトでCommunity Editionとして動作します。commercial水準では、有効なライセンスによってEnterprise Editionが有効にならない限り、プロジェクトサイズに制限のあるEssential Editionとして動作します。詳細および利用水準の変更手順については、[エディション](#)、[利用水準](#)、[ライセンス](#)を参照してください。

Enterprise Editionを利用するには、`license.json` ファイルを次のいずれかの場所に配置します：

SpecForgeは、これらの `license.json` のパスより前に、標準設定ディレクトリとカレントディレクトリの `*-license.json` も検索します。 `SPECFORGE_LICENSE_FILE` を設定した場合は、そのファイルのみを検索します。

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から取得できます。

他のプロバイダー（Gemini、Anthropic、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。

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。

7. Python SDKのインストール（オプション）

Python SDKを使用すると、PythonからプログラムでSpecForgeサーバーと連携できます。PythonノートブックにSpecForgeの解析を組み込んだり、Pandas DataFrameを使ってデータの入出力を直接行ったりすることができます。セットアップ手順については[Python SDKセットアップガイド](#)を参照してください。

参考情報

- [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](#) - specforge.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、以下のLLMプロバイダーをサポートしています：

- **OpenAI**
 - APIキーはplatform.openai.com/api-keysから取得します。
- **Gemini**
 - APIキーはai.google.dev/gemini-api/docs/api-keyから取得します。
- **Anthropic**
 - APIキーはplatform.claude.com/docs/en/api/admin/api_keys/retrieveから取得します。
- **Ollama**
 - docs.ollama.com/quickstartからOllamaをインストールして実行します。

設定方法

VS Code UIでの設定

1. LLMプロバイダーとモデルは、それぞれVS Codeの設定の `specforge.llmProvider` と `specforge.llmModel` から設定できます。
2. VS Codeの[拡張機能設定](#)で `specforge.apiKeySource` 設定が `vscode` に設定されていることを確認してください。
3. コマンドパレットから `SpecForge: Configure API Key for LLM Provider` コマンドを実行します。これにより、選択したプロバイダーのAPIキーの入力を求められます。キーはVS Codeのシークレットストレージに安全に保存されます。

環境変数での設定（Windows、macOS、Linux）

SpecForgeは環境変数を読み取ってLLMプロバイダーを設定します。

1. VS Codeの[拡張機能設定](#)で `specforge.apiKeySource` 設定が `env` に設定されていることを確認してください。
2. 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から取得できます。

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"
```

APIキーはplatform.claude.com/docs/en/api/admin/api_keys/retrieveから取得できます。

Ollama

まず、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。

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
- **Anthropic**：デフォルトは claude-haiku-4-5
- **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サーバーにアクセスする必要があります。接続方法は2つあります：

1. **マネージドサーバー**（specforge.spawnServer）：この設定を有効にすると、拡張機能がSpecForgeサーバープロセスを自動的に起動・管理します。specforge バイナリが PATH にない場合は specforge.serverPath を設定し、ポートを指定する場合は specforge.preferredPort を設定します。
2. **外部サーバー**（デフォルト）：SpecForgeサーバーを自分で起動し（[SpecForgeのセットアップ](#)を参照）、specforge.apiUrl 設定で拡張機能から参照できるようにします（デフォルト：`http://localhost:8080`）。

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。

VSCodeワークスペースは、特定のliloプロジェクトを含むディレクトリにする必要があります。拡張機能は、ワークスペースのルートにある specforge.toml ファイルを使用してプロジェクトの詳細を判断します。プロジェクト構造の例については、[Python SDKセットアップガイド](#)を参照してください。

利用可能なすべての設定（LLM統合、サーバーチューニング、トレースなど）の完全なリファレンスについては、[VSCode拡張機能ガイドの設定セクション](#)を参照してください。

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。

サンプルワークスルー

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。

`uv` を使用する推奨の方法は、プロジェクトごとに個別に `pyproject.toml` ファイルをセットアップすることです。これにより、プロジェクト単位で依存関係を管理し、異なるプロジェクト間の競合を回避できます。典型的なディレクトリ構造は以下のようになります：

```
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)
├── specforge.toml
├── explore.ipynb
├── main.lilo
├── pyproject.toml
└── uv.lock
```

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-0.5.10-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。

- この `.venv` はこのプロジェクト固有のものです。各プロジェクトはそれぞれ独自の `.venv` ディレクトリを持つ必要があります。つまり、プロジェクトごとに個別の `pyproject.toml` ファイルが必要で、プロジェクトごとに個別に `uv sync` を実行しなければなりません。

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
(sample-project) $ which python
/path/to/project/sample-project/.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** `specforge.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。

`specforge.toml` を使用する場合、`[project].source` を指定する必要があります。その他の欠落しているフィールドには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：

- プロジェクト名と明示的なソースパスの設定
- 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
- エディタが表示する解析コードレンズのカスタマイズ
- 反証解析のための `system_falsifier` エントリの登録
- [仕様検索の検証](#)のための `validation_rules` の定義

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). `specforge.toml` が存在する場合は必須。ソースファイルがプロジェクトルートにある場合は "." を使用します。
- `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` — 整合性チェックのSMTソルバータイムアウト:
 - `named` (秒、浮動小数点数). 個々の名前付き仕様ごとのタイムアウト. デフォルト: 0.5
 - `system` (秒、浮動小数点数). システム全体の整合性チェック（すべての仕様を一括チェック）のタイムアウト. デフォルト: 1.0
 - `redundancy.enabled` (ブール値). デフォルト: true
 - `redundancy.timeout` (秒、浮動小数点数). 個々の名前付き仕様ごとのタイムアウト. デフォルト: 0.5
 - `guard_analysis.enabled` (ブール値). デフォルト: true
 - `guard_analysis.timeout` (秒、浮動小数点数). 個々のガード付き仕様ごとのタイムアウト. デフォルト: 0.5
 - `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
- `code_lenses`
 - `satisfiability.enabled` (ブール値). 充足可能性解析のコードレンズを表示します. デフォルト: true

- `satisfiability.timeouts` — 充足可能性コードレンズのSMTソルバータイムアウト:
 - `named` (秒、浮動小数点数). 個々の名前付き仕様ごとのタイムアウト. デフォルト: `diagnostics.consistency.timeouts.named` を継承
 - `system` (秒、浮動小数点数). システム全体の整合性チェックのタイムアウト. デフォルト: `diagnostics.consistency.timeouts.system` を継承
- `redundancy.enabled` (ブール値). 冗長性解析のコードレンズを表示します. デフォルト: `false`
- `redundancy.timeout` (秒、浮動小数点数). 個々の名前付き仕様ごとのタイムアウト. デフォルト: `diagnostics.redundancy.timeout` を継承

省略されたコードレンズのフィールドは、対応する診断設定を継承しますが、`code_lenses.redundancy.enabled` は例外で、デフォルトは `false` です。

- `[[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
- `[[validation_rules]]` (テーブルの配列、オプション)
 - 各エントリ: `antecedent` (クエリ文字列)、`consequent` (クエリ文字列)
 - `--antecedent` / `--consequent` を指定せずに `specforge validate` を実行したときに使用されます。検証を参照してください。
 - 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/"
```

specforge.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
timeout = 0.5

[diagnostics.guard_analysis]
enabled = true
timeout = 0.5

[diagnostics.unused_defs]
enabled = false

[code_lenses.satisfiability]
enabled = true

[code_lenses.redundancy]
enabled = true
timeout = 0.5

[[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"

[[validation_rules]]
antecedent = "label:todo"
consequent = "priority:>50"

[[validation_rules]]
antecedent = "label:blue OR label:green"
consequent = "label:grue"

```

エディションとライセンス

エディション

SpecForgeは、次の3つのエディションのいずれかとして動作します：

エディション	想定される用途	制限
Community	非商用利用	なし
Essential	有効なライセンスを用いない商用利用の評価	システム3個、仕様20個まで
Enterprise	ライセンスに基づく商用利用	なし

エディションは、選択された**利用水準**とSpecForgeが検出したライセンスファイルに基づいて、起動時に決定されます。

- **community**水準では、有効なライセンスが存在する場合でもCommunity Editionとして動作します。
- **commercial**（＝商用）水準では、有効なライセンスがない場合はEssential Edition、ある場合はEnterprise Editionとして動作します。
- 利用水準が選択されていない場合、SpecForgeはデフォルトでCommunity Editionとして動作し、有効なライセンスを検出した場合はEnterprise Editionとして動作します。

利用水準またはライセンスファイルを変更した後は、実行中のSpecForgeプロセスを再起動してください。

利用水準の設定

利用水準（tier）は、SpecForgeを商用で利用するかどうかの指定です。選択できる値は**community**と**commercial**であり、動作するエディションが利用水準とライセンスの状態から決定されます。

利用水準は次のいずれかの方法で設定できます：

- CLI（推奨）
- VS Codeのコマンドパレット
- SpecForgeのグローバル設定ファイル

CLI

次のいずれかを実行します：

```
specforge set-tier community
specforge set-tier commercial
```

VS Code

コマンドパレットを開いて**SpecForge: Set License Tier**を実行し、**Community Edition**または**Commercial Edition**を選択します。

拡張機能自身がサーバーを管理している場合（specforge.spawnServer が有効な場合）、SpecForgeは自動的に再起動します。それ以外の場合は、変更を反映させるためにサーバーを再起動してください。

Liloファイルを開いている間、ステータスバーに現在のエディションが表示されます。クリックすると利用水準を変更できます。

グローバル設定ファイル

CLIおよびVS Codeのコマンドは、SpecForgeのグローバル設定を更新します。次のファイルを直接編集することもできます。

プラットフォーム	グローバル設定
Linuxおよび macOS	\$XDG_CONFIG_HOME/specforge/config.toml、既定では ~/.config/specforge/config.toml
Windows	%APPDATA%\specforge\config.toml

これはプロジェクトの specforge.toml ではありません。トップレベルの tier フィールドに次のいずれかを設定します。

```
tier = "community"
```

```
tier = "commercial"
```

ライセンスファイル

特定の場所のライセンスファイルを使用するには、環境変数 SPECFORGE_LICENSE_FILE にそのパスを設定します。設定しない場合、SpecForgeは次の場所を順に確認し、最初に見つかった有効なライセンスを使用します：

1. グローバル設定ディレクトリ内の *-license.json
2. カレントディレクトリ内の *-license.json
3. グローバル設定ディレクトリ内の license.json
4. カレントディレクトリ内の license.json

ライセンスは、署名、有効期限、失効状態、バージョン互換性のすべての検証に成功した場合に有効となります。検証にネットワーク接続は必要ありません。

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  $\varphi$` ： φ がすべての未来の時刻で真である
- `eventually  $\varphi$` ： φ がある未来の時刻で真である
- `past  $\varphi$` ： φ がある過去の時刻で真であった
- `historically  $\varphi$` ： φ 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]  $\varphi$` は、 φ 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

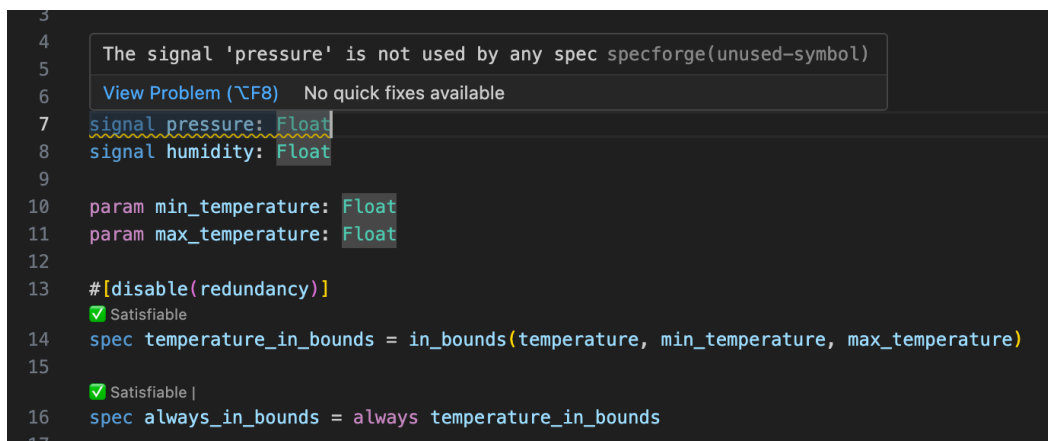
// Humidity should be reasonable when temperature is in normal range
spec humidity_correlation = always (
  (temperature >= 15.0 && temperature <= 35.0) =>
  (humidity >= 20.0 && humidity <= 80.0)
)

// 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。

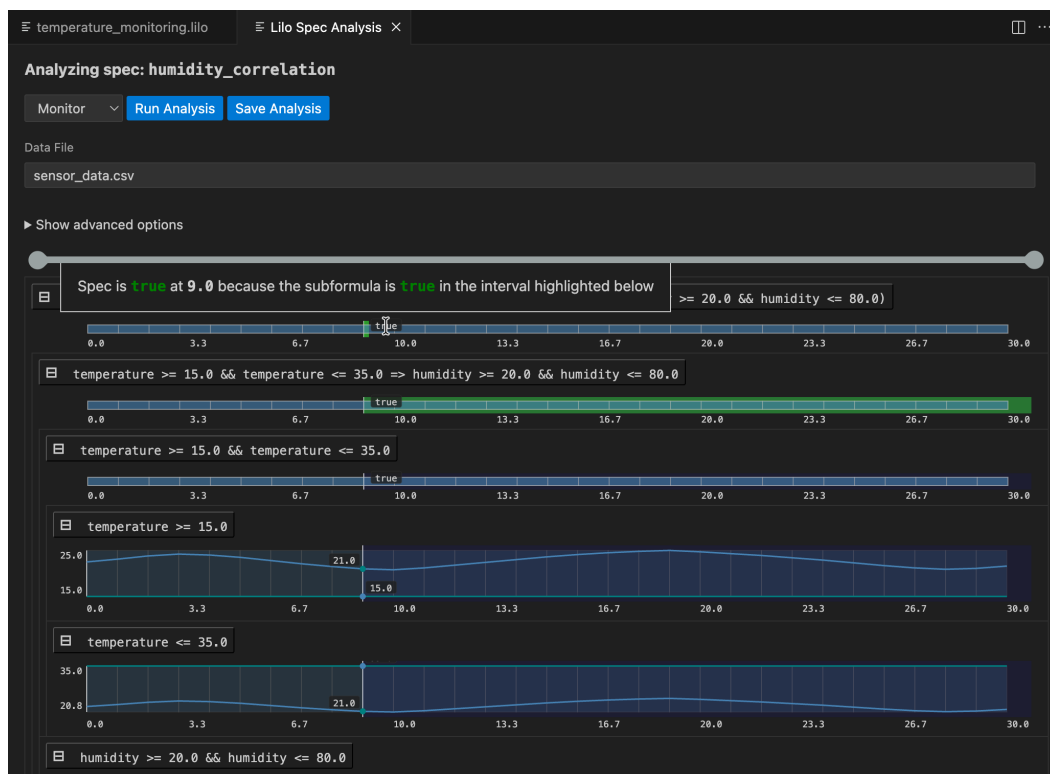
モニタリング

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 Analyze ボタンをクリックします。

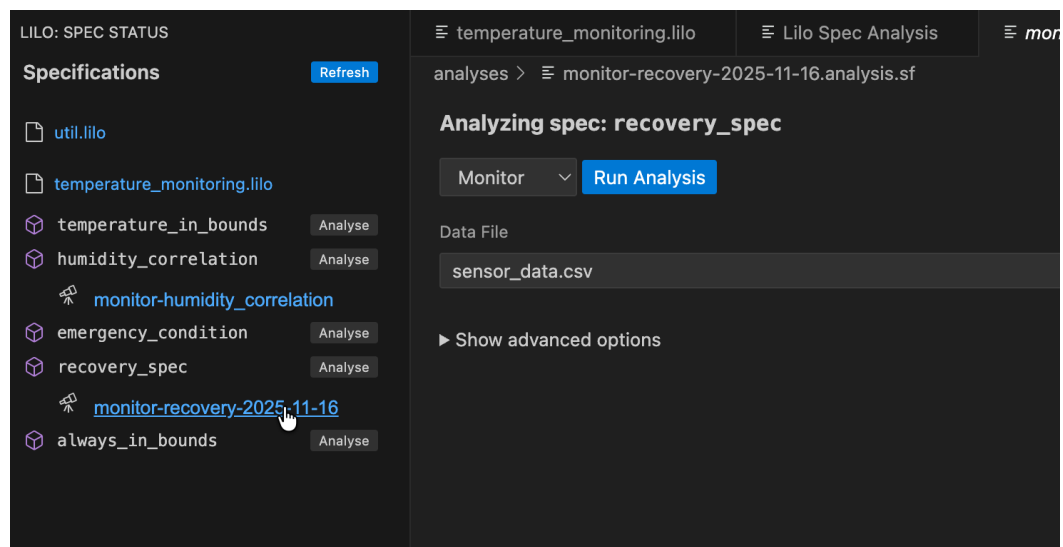
The screenshot shows the SpecForge interface. On the left, the 'Specifications' panel lists several specs: `util.lilo`, `temperature_monitoring.lilo`, `temperature_in_bounds`, `humidity_correlation`, `monitor-humidity_correlation`, `emergency_condition`, `recovery_spec`, and `always_in_bounds`. Each spec has an 'Analyze' button. The `humidity_correlation` button is highlighted with a mouse cursor. On the right, the 'Lilo Spec Analysis' panel shows the analysis results for `temperature_monitoring.lilo`. It displays a list of specs with their status: `Consistent system` (green checkmark), `Satisfiable` (green checkmark), `Satisfiable |` (green checkmark), and `Satisfiable` (green checkmark). The analysis results are shown in a code editor with syntax highlighting.

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。



例示

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



例示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。例示は、仕様を作成する際の補助として使用できます。

反例探索

システムのモデルが利用可能な場合、反例探索を使用して、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、specforge.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。反例探索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。

言語の基礎

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>
```

単位はリテラルの型を `Float` から次元付きの型に変換します。前の例では、型は `Float` から `Float<cm>` または `Float<km/h>` に変換されます。

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)>

単位付きリテラルの演算

単位付きリテラルに対して、加算、減算、比較を行うことができます：

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

これらの演算を行う場合、第1引数と第2引数の単位が同じでなければなりません。そうでない場合、型エラーが発生します。

第1引数と第2引数の単位が異なる場合でも、乗算や除算を行うことができます：

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

これらはそれぞれ 200.0<km*s> および 50.0<km/s> と評価されます。

推論

単位は推論することができます。次の例では、speed の型は Float<m/s> と推論されます。

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

識別

単位は文字列として識別されます。したがって、例えば m と km は単位として無関係です。それらに関連付けるには、例えばkmからmへの変換関数を定義します。

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

単位の宣言

型注釈で使用される単位は `unit` キーワードで宣言する必要があります：

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

宣言された単位は、シグナルと定義の型注釈で使用できます：

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

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：`-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（詳細は[時相演算子の意味論](#)を参照）：
 - `always  $\varphi$` ： `$\varphi$` は将来の全ての時点で真
 - `eventually  $\varphi$` ： `$\varphi$` は将来のある時点で真
 - `past  $\varphi$` ： `$\varphi$` は過去のある時点で真だった
 - `historically  $\varphi$` ： `$\varphi$` は過去の全ての時点で真だった
 - `will_change  $\varphi$` ： `$\varphi$` は将来のある時点で値が変わる
 - `did_change  $\varphi$` ： `$\varphi$` は過去のある時点で値が変わった
 - `$\varphi$  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 `$\varphi$` が真だった
 - `$\varphi$  until  $\psi$` ： `$\varphi$` が真になるまでの将来の全ての時点で `$\varphi$` が真
 - `$\varphi$  releases  $\psi$` ：`!( $\varphi$  until  $\psi$ )` の省略形。
 - `next f`：1つ次の（離散的な）時点における `f`（必ずしもブール値でなくてもよい）の値。1つ次の時点が存在しない場合は値が保持される。
 - `previous f`：1つ前の（離散的な）時点における `f`（必ずしもブール値でなくてもよい）の値。1つ前の時点が存在しない場合は値が保持される。

単項の時相演算子は括弧なしで連鎖させることができます。例えば、`always eventually (x < 0)` のように書けます。

時相演算子は区間で修飾できます：

- `always [a, b]  $\varphi$` ：`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 `$\varphi$` が真
- `eventually [a, b]  $\varphi$` ：`a` 時間単位先から `b` 時間単位先までの間のある時点で `$\varphi$` が真
- `$\varphi$  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 `$\varphi$` が真
- `$\varphi$  releases [a, b]  $\psi$` ：`!( $\varphi$  until [a, b]  $\psi$ )` の省略形。
- 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]`。
- スライディングウィンドウ演算子
 - `max_future [0, a] ψ`：次の `a` 時間単位における `ψ` の最大値。
 - `min_future [0, a] ψ`：次の `a` 時間単位における `ψ` の最小値。
 - `max_past [0, a] ψ`：過去の `a` 時間単位における `ψ` の最大値。
 - `min_past [0, a] ψ`：過去の `a` 時間単位における `ψ` の最小値。
 - 区間は `[0, a]` の形式でなければなりません。
 - 他の時相演算子と同様に区間は省略可能で、その場合は `[0, infinity]` を意味します。
- 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、`-(-x)` や `!(next ψ)` 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。
- `sqr` は数値の平方根を返します。引数は `Int` または無次元の `Float` でなければなりません：

```
sqr(x)
```

- `abs` は数値の絶対値を返します。引数は `Int` または `Float` でなければならず、単位は保持されます：

```
abs(x)
```

- `max` は2つ以上の数値の最大値を返します。すべての引数は同じ型と単位を持つ必要があります：

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

- `min` は2つ以上の数値の最小値を返します。すべての引数は同じ型と単位を持つ必要があります：

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

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。

Case式

条件式のすべての分岐先の式が `Bool` 型の場合、 `cases` 式を使用できます。

```
cases {
  temp > 30.0 -> eventually temp < 20.0;
  temp < 10.0 -> eventually temp > 20.0;
  10.0 <= temp <= 30.0 -> true;
}
```

これは $(temp > 30.0 \Rightarrow \text{eventually } temp < 20.0) \ \&\& \ (temp < 10.0 \Rightarrow \text{eventually } temp > 20.0) \ \&\& \ (10.0 \leq temp \leq 30.0 \Rightarrow true)$ と解釈されます。したがって、 `10.0 <= temp <= 30.0 -> true` のケースは省略できます。ただし、わかりやすさのために、網羅的かつ互いに排反な条件を記述することを推奨します。

レコード型

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
```

列挙型

Liloは（nominalな）列挙型をサポートしています。列挙型はペイロードを含むことはできません。enum キーワードに続けて名前と空でないコンストラクタの集合を記述して宣言します。コンストラクタは # 記号で始まる必要があります。

例：

```
enum Color = #Red | #Green | #Blue
```

列挙型は独自の名前空間を作成し、デフォルトで開いています。複数の列挙型が同一のコンストラクタ名を共有することもできます。コンストラクタは名前のみ、または完全修飾名で参照できます：

```
#Red
#Color::Red
#Utils::Color::Red
```

コンストラクタは等価比較をサポートしています：

```
def is_red(c: Color): Bool = c == #Red
```

データ内では、列挙型のコンストラクタは先頭に # を付けて記述し、引用符や修飾は付けません。詳細については、[データファイル](#)に関する章を参照してください。

パターンマッチ

パターンマッチは現在、列挙型のみをサポートしています。match キーワードに続けて被検査式と網羅的なケースの集合を記述します。コンストラクタを参照する際の規則は、ケースにも同じく適用されます。

例：

```
def is_green(c: Color): Bool =
  match c {
    #Red -> false;
    #Color::Green -> true;
    #Utils::Color::Blue -> false;
  }
```

パターンマッチでは、複数の列挙型で使われているコンストラクタ名の曖昧性も、推論により自動で解消されます：

```
enum Mode = #On | #Off
enum Switch = #On | #Off

def f(x: Mode): Int = match x { #On -> 1; #Off -> 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

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可能です。

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する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ています。

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)
```

仮定

`assumption` は、システムを解析する際に所与のものとして扱われるプロパティを宣言します。構文的には `spec` と同様で、パラメータを持たず戻り値の型は常に `Bool` ですが、ツールによる扱いが異なります：仮定は検証対象ではなく、例示や充足可能性チェックの際に制約として自動的に含まれます。

```
signal temperature: Float  
signal heater_on: Bool  
  
assumption physics = always (heater_on => next temperature >= temperature)  
  
spec eventually_warm = eventually (temperature > 30.0)
```

この例では、SpecForgeがトレース例を生成する際、または仕様の充足可能性をチェックする際に、`physics` は必ず成り立たなければなりません。仮定と解析の相互作用の詳細については、[例示と充足可](#)

[能性](#)を参照してください。

モジュール

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
```

別モジュールから単位をインポートするには、`use` リストの各単位に `unit` キーワードをプレフィックスとして付けます：

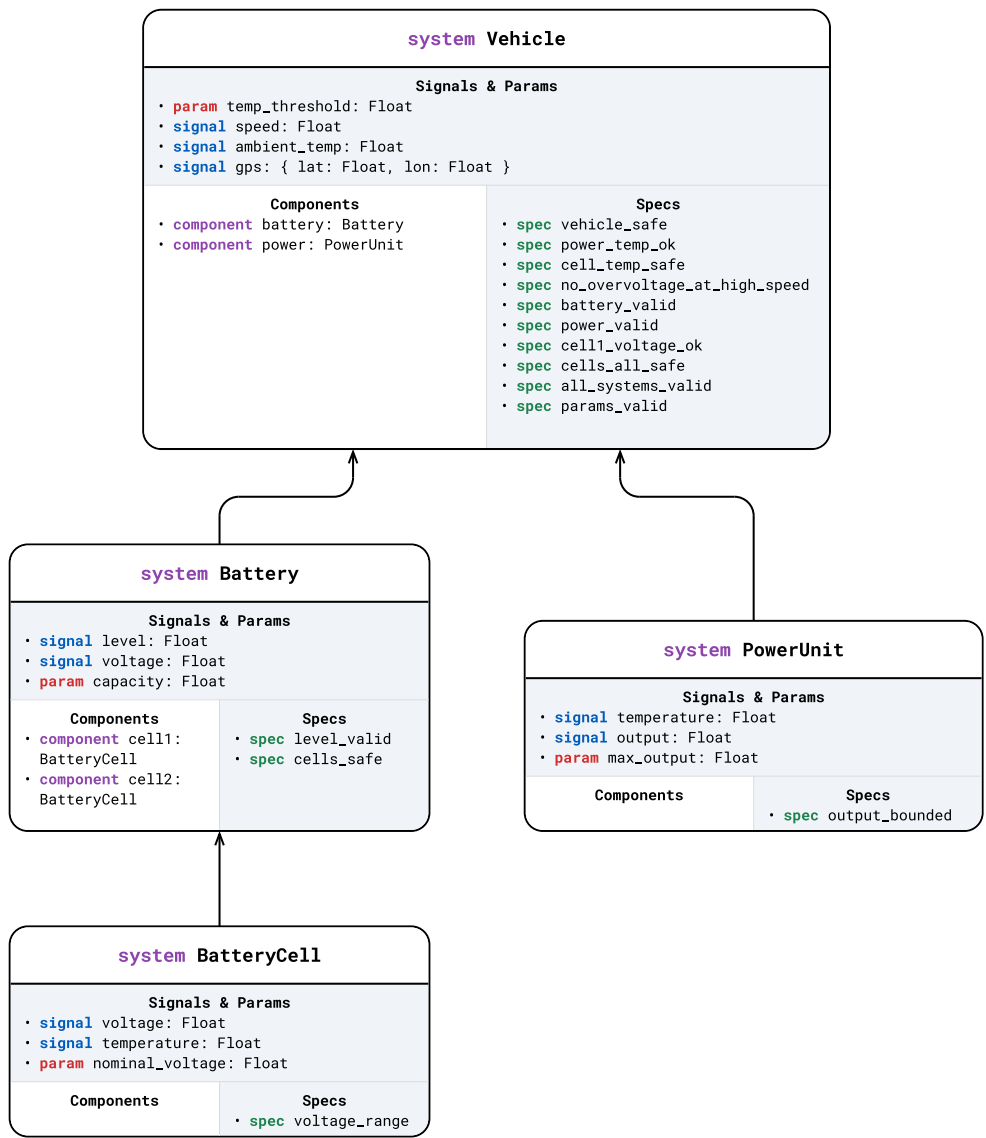
```
import Units use { unit m, unit s }
```

コンポーネント

コンポーネントは、Lilo のシステムを階層的な構成要素として組織化するための仕組みです。

次のように構成された Vehicle システムを考えてみましょう：

- Vehicle は PowerUnit と Battery をコンポーネントとして持ちます。
- バッテリーは2つの BatteryCell から構成されます。
- PowerUnit、Battery、BatteryCell、Vehicle の各システムは、それぞれ独自のシグナル、パラメータ、仕様を持ちます。



BatteryCell と PowerUnit はコンポーネントを持たないため、次のように単純に宣言できます：

```
// 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
```

ここで pub キーワードは、これらの signal、param、def が、これらのシステムをコンポーネントとしてインスタンス化する他のシステムからアクセス可能であることを示します。spec で宣言された仕様は常に公開されます。

コンポーネントのインスタンス化

Battery システムは BatteryCell コンポーネントを使用します。これは component キーワードを使って宣言できます：

```
// Battery.lilo

system Battery

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

component cell1: BatteryCell

// ...
```

リフトされるシグナル・パラメータとマップされるシグナル・パラメータ

あるシステムが別のシステムのコンポーネントとしてインスタンス化されると、そのパラメータとシグナルは親システムへリフトされます。したがって、Battery のスキーマには level、voltage、capacity に加えて、cell1::voltage、cell1::temperature、cell1::nominal_voltage が含まれます。

コンポーネントのシグナルやパラメータの一部は**マップ**されることがあります。これは、それらが入力スキーマの一部ではなく、その値が（親システムの）他のシグナルやパラメータによって決定されることを意味します。例えば次の例では、Battery のスキーマには依然として cell1::voltage、cell1::temperature、cell1::nominal_voltage、および cell2::temperature が現れますが、cell2::nominal_voltage や cell2::voltage は現れません。cell2::nominal_voltage の値は 3.7 に固定され、cell2::voltage の値は Battery の voltage シグナルから導出されます：

```
// 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
}
```

コンポーネントのシグナルとパラメータへのアクセス

システム内の Lilo 式では、`::` 演算子を使ってコンポーネントのシグナルやパラメータを参照できます：

```
// 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
```

システムは、そのコンポーネントが持つコンポーネントのシグナルやパラメータも参照できます。例えば、Battery コンポーネントと PowerUnit コンポーネントの両方を持つ Vehicle システムを考えます。Vehicle システムでは、次のように仕様を記述できます：

```
// 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)

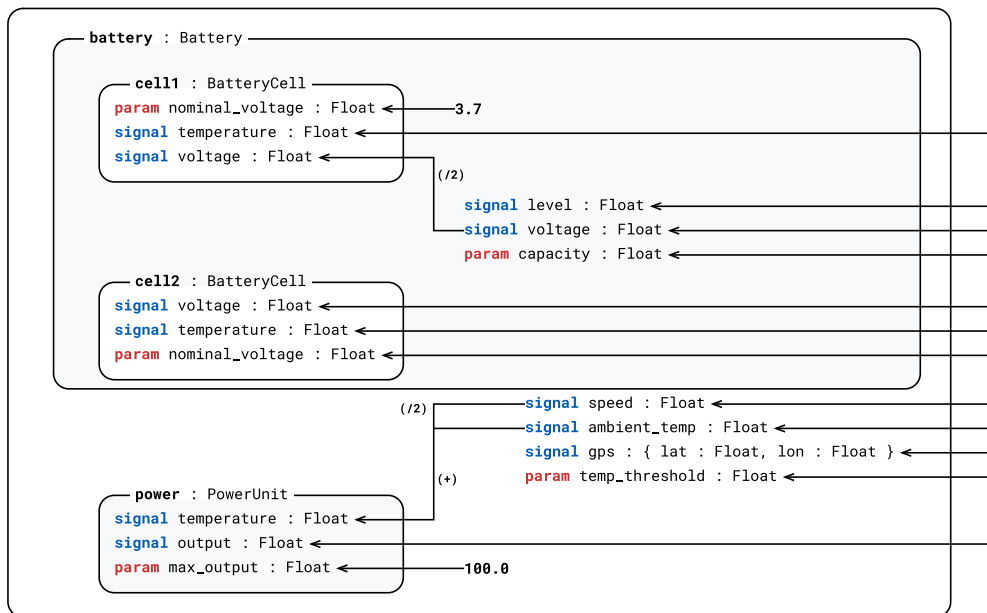
def any_cell_overvoltage: Bool = battery::cell1::over_voltage || battery::cell2::over_voltage

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

// ...
```

システムスキーマ

システムのスキーマとは、そのシステムの入力を構成するシグナルとパラメータの集合を指します。モニタリングを行う際、ユーザーはスキーマの各要素に対して値を与えます。Vehicle システムに対応するスキーマを下図に示します：



システムのスキーマは、[SpecForge CLI](#) の `schema` コマンドで確認できます。マップされたシグナルやパラメータは、その値が別のシグナルやパラメータから導出されるため、スキーマには現れないことに注意してください：

```
$ 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)
```

`schema` コマンドは `--diff` オプションを付けて実行することで、Lilo システムのスキーマをパラメータファイルやデータファイルと比較できます。このコマンドは、ファイル内で不足しているシグナル・パラメータや余分なシグナル・パラメータ、および型の不一致を報告します：

```
$ specforge schema --system Vehicle --only signals --diff --datafile vehicle_data.json
Signal Diff Summary:
  2 missing, 1 mismatched, 1 expected-scalars, 2 extra

Missing Fields:
  battery::voltage
  battery::cell2::temperature

Mismatched Types:
  speed expected Float but got String

Expected Scalars but Found Records Instead:
  ambient_temp

Extra Fields:
  ambient_temp.value
  battery.cell2.temperure (did you mean battery::cell2::temperature?)
```

モニタリング時にスキーマへ値を与える際の形式の詳細については、[データファイル](#)のドキュメントを参照してください。

静的解析

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。

ケース式のガード解析

仕様がケース式である場合、ケース式内のガードについて充足可能性検査、網羅性検査、および排他性検査を行います。

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

- 充足可能性検査: 各ガードが単独で充足可能かどうか。
- 網羅性検査: ガード全体がすべての可能なケースを網羅しているかどうか。

- 排他性検査: ガードが互いに排他的かどうか。

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。
- ソフト仮定： `#[rigidity = "soft"]`
 - `assumption` に適用すると、ソフト制約としてマークされます。ソルバーはこの制約を満たすように試みますが、ハード制約と矛盾する場合は緩和することがあります。詳細は[厳格度レベル](#)を参照してください。

ラベル、エイリアス、カスタムフィールド

属性を使用することで、仕様に対してラベル、エイリアス、カスタムフィールドで注釈を付けることができます。

シグナルやパラメータのエイリアスは、データファイル内でそれらを指し示すために使うことができます。詳細は[データファイル](#)を参照してください。

エイリアスは一意であることが期待されます。すなわち、あるエイリアスが別のエイリアスや他の宣言の名前と重複してはなりません。

ラベルの色は `specforge.toml` 設定ファイルの `[labels.colors]` セクションで設定できます。色は16進コードまたは標準的なHTML5の色名で表されます。

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

カスタムフィールドを使用することで、仕様に追加のスカラーメタデータを付加できます。レビューステータス、担当者、優先度など、プロジェクト固有の分類に役立ちます。

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

カスタムフィールドの値はスカラー値（文字列、数値、ブール値）でなければなりません。単一の仕様で同じカスタムフィールドが複数回指定された場合、最初の値が使用され、警告が出力されます。

VSCoDe拡張機能では、仕様ステータスペインでのフィルタリングにカスタムフィールドを使用できます。例：

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

完全なクエリ構文については、[VSCoDe拡張機能](#)を参照してください。

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、例示機能は、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、これによりSpecForgeは該当パラメータのデフォルト値を無視ようになります。

```
system main

signal p: Int
param bound: Int = 1

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

- 例示
 - `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、通常の定義と同様に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
```

スタブは、まだ形式的に与えられていない[仮定](#)にも使用できます。

```
/// height は常に非負である  
assumption height_non_negative
```

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  
}
```

時相演算子の意味論

Liloは、離散的な時間と連続的な時間の両方を取り扱うという意味で、やや非標準的な時相論理の意味論を採用しています。

背景

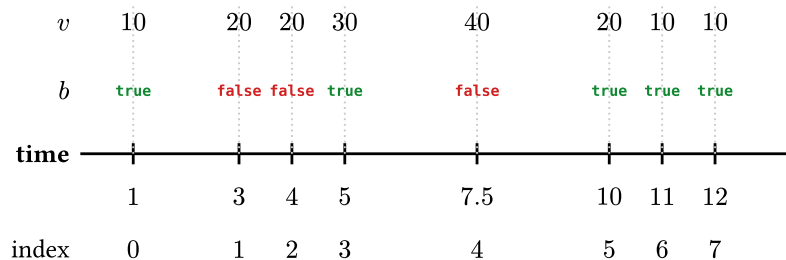
T を型とします。型 T の**サンプル**とは、 $\{\text{time} : \text{Real}, \text{value} : T\}$ というペアです。型 T の**信号**とは、時間値が狭義単調増加である型 T のサンプルが少なくとも2つ並んだ有限列です。 $\text{Signal}[T]$ を型 T のすべての信号の集合と記します。信号の最初のサンプルが時刻 0 である必要はありません。信号 σ が与えられたとき、 σ の要素を $\sigma[0]$ 、 $\sigma[1]$ 、...、 $\sigma[n-1]$ と表記します (n は σ のサンプル数)。 σ のサンプル数と σ の持続時間 ($\sigma[n-1].\text{time} - \sigma[0].\text{time}$) は区別することに注意してください。

信号 σ の**サポート** ($\text{support}(\sigma)$ と表記) は、 σ のサンプルに紐づいた時間値の集合、すなわち $\text{support}(\sigma) = \{\sigma[i].\text{time} \mid 0 \leq i < n\}$ です。信号のサンプルにインデックスを付けるため、2つの異なる記法を使用します。時刻 t における σ の値を $\sigma[\text{time} = t]$ と記します。すなわち $\sigma[\text{time} = t] = \sigma[i].\text{value}$ (ここで i は $\sigma[i].\text{time} = t$ となるインデックス)。 $\sigma[\text{time} = t]$ は $t \in \text{support}(\sigma)$ の場合にのみ定義されることに注意してください。

例として、次のサンプルによって定義される信号 $v : \text{Int}$ と $b : \text{Bool}$ を考えます：

```
v = [{time: 1.0, value: 10}, {time: 3.0, value: 20}, {time: 4.0, value: 20}, {time: 5.0, value: 30}, {time: 7.5, value: 40}, {time: 10.0, value: 20}, {time: 11.0, value: 10}, {time: 12.0, value: 10}]
b = [{time: 1.0, value: true}, {time: 3.0, value: false}, {time: 4.0, value: false}, {time: 5.0, value: true}, {time: 7.5, value: false}, {time: 10.0, value: true}, {time: 11.0, value: true}, {time: 12.0, value: true}]
```

これらは次のように可視化できます。



v と b にはどちらも 0 から 7 までの番号が付いた8つのサンプルがあります。両方の信号のサポートは $\{1.0, 3.0, 4.0, 5.0, 7.5, 10.0, 11.0, 12.0\}$ です。値 30 を持つ同じサンプルを $v[\text{time} = 5.0]$ または $v[3]$ として参照できます。

各Lilo式 φ は、Liloの型システムによって何らかの型 T を持つと判定されます ($\varphi : T$ と記します)。型 T のLilo式の意味論は、 $\text{Signal}[S] \rightarrow \text{Signal}[T]$ 型の関数として理解されます (S はシステムに記述された入力信号の型)。この関数は**サポートを保存**します。すなわち、出力信号は入力信号とまったく同じ時間値上で定義されます。型 T の式 φ に対して、 $\llbracket \varphi \rrbracket$ でこの関数を表します。

非時相演算子には、標準的なブール演算子、算術演算子、比較演算子、条件分岐などが含まれます。ある式が**ポイントワイズ**な式であるとは、任意の固定された時間値 t に対して、出力信号 $\llbracket \varphi \rrbracket(\sigma)[\text{time} = t]$ の値が入力信号の値 $\sigma[\text{time} = t]$ のみに依存することをいいます。Lilo式が**非時相的**であるとは、その解釈が入力信号に依存せず、したがって時間の経過にかかわらず一定であることをいいます。つまり、ポイントワイズな式は入力信号と非時相演算子から構成され、非時相式はパラメータ、定数、および非時相演算子から構成されます。

次のLiloシステムを考えます。

```

system main

signal v : Int
signal b : Bool
param p : Int

```

ここで、式 $v * 2$ はポイントワイズであり、式 $p * 2$ は非時相的です。式 $\text{eventually } [0, 5] (v > p)$ はポイントワイズでも非時相的でもありません。

実数の区間は $[a, b]$ というペアで書きます ($a: \text{Real}$ 、 $b: \text{Real} \mid \text{infinity}$ 、 $0 \leq a \leq b$)。 b が infinity である場合、その区間は非有界であるといい、そうでない場合は有界であるといいます。時間点 t と区間 I に対して、 $t + I$ または $t - I$ でそれぞれ $\{t + x \mid x \in I\}$ または $\{t - x \mid x \in I\}$ を表します。

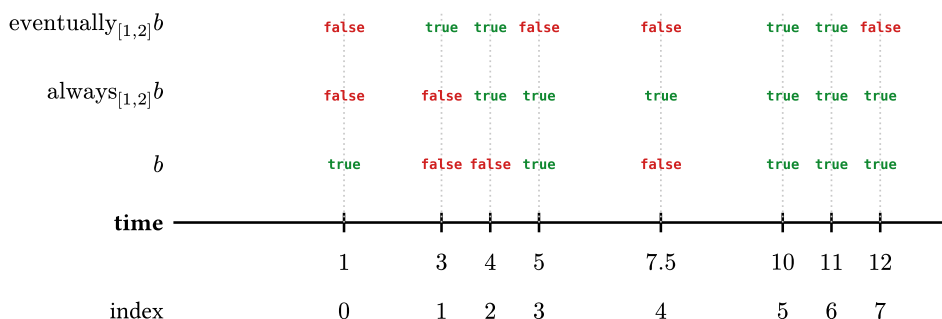
区間 $t + I$ または $t - I$ は、入力信号のサポートから完全または部分的に外れ、その上限や下限を超える場合があります。しかし、時相演算子はサポートを保存し、これらの区間とサポートとの共通部分を使って定義されるため、問題にはなりません。

時相演算子とその意味論

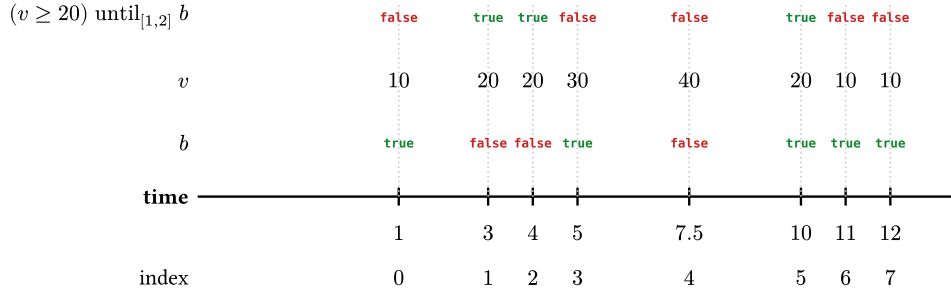
Liloの時相演算子は、一部の例外を除いて区間で注釈されます。 $\varphi : \text{Bool}$ が与えられたとき、以下の時相演算子を定義します。

- **Always:** すべての $t' \in (t + I) \cap \text{support}(\sigma)$ に対して $\llbracket \varphi \rrbracket (\sigma)[\text{time} = t'] = \text{true}$ である場合、 $\llbracket \text{always}[I] \varphi \rrbracket (\sigma)[\text{time} = t] = \text{true}$ です。
- **Eventually:** $\llbracket \varphi \rrbracket (\sigma)[\text{time} = t'] = \text{true}$ となる $t' \in (t + I) \cap \text{support}(\sigma)$ が存在する場合、 $\llbracket \text{eventually}[I] \varphi \rrbracket (\sigma)[\text{time} = t] = \text{true}$ です。
- **Historically:** すべての $t' \in (t - I) \cap \text{support}(\sigma)$ に対して $\llbracket \varphi \rrbracket (\sigma)[\text{time} = t'] = \text{true}$ である場合、 $\llbracket \text{historically}[I] \varphi \rrbracket (\sigma)[\text{time} = t] = \text{true}$ です。
- **Past:** $\llbracket \varphi \rrbracket (\sigma)[\text{time} = t'] = \text{true}$ となる $t' \in (t - I) \cap \text{support}(\sigma)$ が存在する場合、 $\llbracket \text{past}[I] \varphi \rrbracket (\sigma)[\text{time} = t] = \text{true}$ です。
- **Until:** $\llbracket \varphi \rrbracket (\sigma)[\text{time} = t'] = \text{true}$ となる $t' \in (t + I) \cap \text{support}(\sigma)$ が存在し、かつ $t \leq t'' < t'$ を満たすすべての $t'' \in \text{support}(\sigma)$ に対して $\llbracket \varphi \rrbracket (\sigma)[\text{time} = t''] = \text{true}$ である場合、 $\llbracket \varphi \text{ until}[I] \varphi \rrbracket (\sigma)[\text{time} = t] = \text{true}$ です。
- **Since:** $\llbracket \varphi \rrbracket (\sigma)[\text{time} = t'] = \text{true}$ となる $t' \in (t - I) \cap \text{support}(\sigma)$ が存在し、かつ $t' < t'' \leq t$ を満たすすべての $t'' \in \text{support}(\sigma)$ に対して $\llbracket \varphi \rrbracket (\sigma)[\text{time} = t''] = \text{true}$ である場合、 $\llbracket \varphi \text{ since}[I] \varphi \rrbracket (\sigma)[\text{time} = t] = \text{true}$ です。

上記の例の信号 b について、式 $\text{always } [1,2] b$ と $\text{eventually } [1,2] b$ は次のように評価されます。



式 $(v \geq 20) \text{ until } [1, 2] b$ は次のように評価されます。



`always` と `eventually` は互いに論理的双対であることに注意してください。 `until` の論理的双対は `releases` であり、 $\varphi \text{ releases}[I] \psi$ は $\neg(\neg\varphi \text{ until}[I] \neg\psi)$ として定義されます。演算子 `since`、`historically`、`past` はそれぞれ `until`、`always`、`eventually` の過去時相版です。

`will_change` と `did_change` は、等号が定義されている任意の適切に型付けされた $\varphi : T$ （実際には、すべての非関数型）に対して次のように定義されます。

- **Will Change:** $\llbracket \varphi \rrbracket(\sigma)[\text{time} = t'] \neq \llbracket \varphi \rrbracket(\sigma)[\text{time} = t'']$ となる2点 $t', t'' \in (t + I) \cap \text{support}(\sigma)$ が存在する場合、 $\llbracket \text{will_change}[I] \varphi \rrbracket(\sigma)[\text{time} = t] = \text{true}$ です。
- **Did Change:** $\llbracket \varphi \rrbracket(\sigma)[\text{time} = t'] \neq \llbracket \varphi \rrbracket(\sigma)[\text{time} = t'']$ となる2点 $t', t'' \in (t - I) \cap \text{support}(\sigma)$ が存在する場合、 $\llbracket \text{did_change}[I] \varphi \rrbracket(\sigma)[\text{time} = t] = \text{true}$ です。

以下の演算子には区間を付けることができません。また重要なこととして、これらは離散時間意味論で定義されます。すなわち、関連付けられた `time` ではなく、サンプルに関連付けられた離散的なインデックスに依存します。基本となる式 φ の型が T である場合、式 `next  $\varphi$` 、`previous  $\varphi$` も同様に型 T を持ちます。 n を入力信号 σ のサンプル数とします。

- **Next:** $0 \leq i < n - 1$ に対して $\llbracket \text{next } \varphi \rrbracket(\sigma)[i] = \llbracket \varphi \rrbracket(\sigma)[i + 1]$ 、 $\llbracket \text{next } \varphi \rrbracket(\sigma)[n - 1] = \llbracket \varphi \rrbracket(\sigma)[n - 1]$ 。
- **Previous:** $0 < i < n$ に対して $\llbracket \text{previous } \varphi \rrbracket(\sigma)[i] = \llbracket \varphi \rrbracket(\sigma)[i - 1]$ 、 $\llbracket \text{previous } \varphi \rrbracket(\sigma)[0] = \llbracket \varphi \rrbracket(\sigma)[0]$ 。

以下のスライディングウィンドウ演算子は、値が数値型である任意の式 φ に対して定義されます。また、関連する区間は $[0, b]$ の形式（ b は `infinity` を含む）である必要があります。

- **Future Maximum:** $\llbracket \text{max_future } [0, a] \varphi \rrbracket(\sigma)[\text{time} = t] = \max\{\llbracket \varphi \rrbracket(\sigma)[\text{time} = t'] \mid t' \in \text{support}(\sigma), t \leq t' \leq t + a\}$ 。
- **Past Maximum:** $\llbracket \text{max_past } [0, a] \varphi \rrbracket(\sigma)[\text{time} = t] = \max\{\llbracket \varphi \rrbracket(\sigma)[\text{time} = t'] \mid t' \in \text{support}(\sigma), t - a \leq t' \leq t\}$ 。
- **Future Minimum:** $\llbracket \text{min_future } [0, a] \varphi \rrbracket(\sigma)[\text{time} = t] = \min\{\llbracket \varphi \rrbracket(\sigma)[\text{time} = t'] \mid t' \in \text{support}(\sigma), t \leq t' \leq t + a\}$ 。
- **Past Minimum:** $\llbracket \text{min_past } [0, a] \varphi \rrbracket(\sigma)[\text{time} = t] = \min\{\llbracket \varphi \rrbracket(\sigma)[\text{time} = t'] \mid t' \in \text{support}(\sigma), t - a \leq t' \leq t\}$ 。

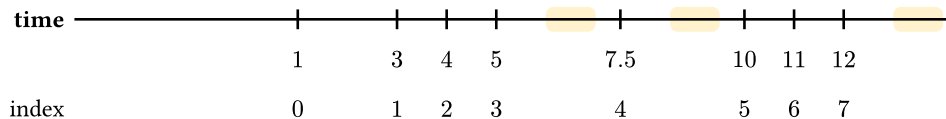
信号の境界における時相演算子の動作

ある $\text{time} = t$ において、区間 $t + I$ または $t - I$ が入力信号 σ のサポートから完全または部分的に外れることがあります。このような場合の時相演算子の動作を明確にします。

- **Eventually, Past:** `eventually[I]  $\varphi$` または `past[I]  $\varphi$` が真になるには、 φ が真となる証拠が存在しなければなりません。具体的には、 $(t + I) \cap \text{support}(\sigma) = \emptyset$ または $(t - I) \cap \text{support}(\sigma) = \emptyset$ の場合、 $\llbracket \text{eventually}[I] \varphi \rrbracket(\sigma)[\text{time} = t] = \text{false}$ および $\llbracket \text{past}[I] \varphi \rrbracket(\sigma)[\text{time} = t] = \text{false}$ です。
- **Always, Historically:** 式 `always[I]  $\varphi$` または `historically[I]  $\varphi$` は、対象となる区間とサポートとの共通部分のすべての点における φ の真偽によって決まります。この共通部分が空の場合、式は空虚に真となります。形式的には、 $(t + I) \cap \text{support}(\sigma) = \emptyset$ または $(t - I) \cap \text{support}(\sigma) = \emptyset$ の場合、 $\llbracket \text{always}[I] \varphi \rrbracket(\sigma)[\text{time} = t] = \text{true}$ および $\llbracket \text{historically}[I] \varphi \rrbracket(\sigma)[\text{time} = t] = \text{true}$ です。

- **Until, Since:** 式 $\psi \text{ until}[I] \varphi$ または $\psi \text{ since}[I] \varphi$ の場合、 ψ の証拠が存在しなければなりません。 $(t + I) \cap \text{support}(\sigma) = \emptyset$ または $(t - I) \cap \text{support}(\sigma) = \emptyset$ の場合、 $\llbracket \psi \text{ until}[I] \varphi \rrbracket(\sigma)[\text{time} = t] = \text{false}$ および $\llbracket \psi \text{ since}[I] \varphi \rrbracket(\sigma)[\text{time} = t] = \text{false}$ です。一方、 t と ψ の証拠との間に $\text{support}(\sigma)$ の時間点が存在しない場合、式はどの点でも φ が真であることを要求しません。
- **Next, Previous:** $\text{next } \varphi$ と $\text{previous } \varphi$ の値は、対象となる方向で最も近いサンプルによって定義されます。 $\llbracket \text{previous } \varphi \rrbracket(\sigma)[0] = \llbracket \varphi \rrbracket(\sigma)[0]$ です。信号に n 個のサンプルがある場合、最後のサンプルについて $\llbracket \text{next } \varphi \rrbracket(\sigma)[n - 1] = \llbracket \varphi \rrbracket(\sigma)[n - 1]$ が成り立ちます。
- **Will Change, Did Change:** 式 $\text{will_change}[I] \varphi$ と $\text{did_change}[I] \varphi$ は、対象となる区間とサポートとの共通部分に2つの異なる時間点があることを要求します。 $(t + I) \cap \text{support}(\sigma)$ または $(t - I) \cap \text{support}(\sigma)$ に時間点が1つしかない、あるいはまったくない場合、 $\llbracket \text{will_change}[I] \varphi \rrbracket(\sigma)[\text{time} = t] = \text{false}$ および $\llbracket \text{did_change}[I] \varphi \rrbracket(\sigma)[\text{time} = t] = \text{false}$ です。
- **Maximum, Minimum:** 式 $\{\text{max}, \text{min}\}_{\{\text{future}, \text{past}\}}$ で許可される区間は 0 から始まる必要があります。したがって、 $t \in \text{support}(\sigma)$ であれば $t \in ((t + I) \cap \text{support}(\sigma))$ も成り立つため、この集合が空になることはありません。この集合が単集合の場合、式は時刻 t 自体における φ の値に評価されます。

上図の式 $\text{always } [1, 2] b$ と $\text{eventually } [1, 2] b$ の値を考えます。時刻 $t = 5, 7.5, 12$ では、区間 $t + [1, 2]$ はそれぞれ $[6, 7]$ 、 $[8.5, 9.5]$ 、 $[13, 14]$ です。下のタイムラインに示すように、これらの区間と b のサポートとの共通部分は空です。したがって eventually の証拠は存在せず、式 $\text{eventually } [1, 2] b$ はこれらすべての点で false に評価されます。一方、式 $\text{always } [1, 2] b$ は空虚に真となるため、これらすべての点で true に評価されます（つまり、この式が制約を課す b のサポート上の点がありません）。



同様に、式 $(v \geq 20) \text{ until } [1, 2] b$ は、時刻 $t = 5$ 、 $t = 7.5$ 、 $t = 12$ において、対象となる区間に b の証拠がないため false に評価されます。

例示と充足可能性

例示とは、仕様を満たすトレースの例を生成するプロセスです。

充足可能性は例示と密接に関連する概念で、仕様を真にするトレースが少なくとも1つ存在する場合、その仕様は充足可能です。SpecForgeの**静的解析**は、仕様が充足可能かどうかを自動的にチェックし、充足不可能な場合は警告を表示します。

例示と充足可能性チェックはどちらも、ソルバーが考慮する追加の制約である**仮定**の影響を受けます。

仮定

仮定とは、仕様を解析する際に所与のものとして扱われるプロパティです。検証したいものではなく、物理法則、環境の制約、または成立することが期待される動作条件など、システムに関する背景知識を表します。

assumptionキーワード

Liloでは、`assumption` キーワードを使って仮定を宣言できます：

```
signal temperature: Float
signal heater_on: Bool

assumption physics = always (heater_on => next temperature >= temperature)

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

`assumption` は構文的には `spec`（パラメータを持たず、戻り値の型は常に `Bool`）と同様ですが、ツールによる扱いが異なります：

- **例示**：同じシステム内のすべての仕様のトレース例を生成する際、すべての仮定は自動的に制約として含まれます。ソルバーは仕様とすべての仮定の両方を満たすトレースを生成する必要があります。
- **充足可能性**：仕様が充足可能かどうかをチェックする際、仮定が制約セットに含まれます。単独では充足可能な仕様も、仮定のもとでは充足不可能になる場合があります。
- **冗長性**：仮定は冗長性チェックの際に他の仕様と並べて含まれます。仮定（および他の仕様）から導出される仕様は冗長として警告されます。

仮定に `#[rigidity = "soft"]` **属性**を付けることで、その仮定を例示器に対するソフト制約にすることができます。詳細は下記の**厳格度レベル**のセクションを参照してください。デフォルトでは、仮定はハードとして扱われます。

したがって、解析を実行する際に仮定を手動で渡す必要はありません。Liloソースで宣言するだけで十分です。

アドホック仮定

`assumption` キーワードに加えて、**Python SDK**や**VSCoDe拡張機能**を通じて例示を実行する際に**アドホック仮定**を渡すこともできます。これらは、システム定義の一部ではなく、特定の解析実行にのみ適用される一時的な制約です。

アドホック仮定は、Liloソースを変更せずに「もし〜だったら」というシナリオを探索したい場合に便利です。例えば、「温度が20度を超えた状態から始まるトレースを生成する」といった場合に使用します。

厳格度レベル

ソース中で `assumption` キーワードを使って宣言されたもの、例示クエリにアドホックに渡されたものの別にかかわらず、各仮定にはソルバーがどのように扱うかを制御する**厳格度**レベルがあります：

- **Hard (ハード)**：仮定は絶対的な制約として扱われます。生成されたトレースは仮定を**必ず**満たさなければなりません。仕様とすべてのハード仮定の両方を満たすトレースを見つけられない場合、例示は失敗します。
- **Soft (ソフト)**：仮定は優先条件として扱われます。ソルバーはまず仕様とすべてのハード仮定を満たすトレースを見つけようとし、次にソフト仮定も満たすように試みます。ソフト仮定がハード制約と共に満たせない場合、それは緩和され、ソルバーはそれでも結果を生成します。

実際には、必須の制約（物理法則、安全境界など）には**Hard**を使用し、必須ではないが望ましい制約（「温度はある時点で上昇すべき」など）には**Soft**を使用してください。

注意： `assumption` キーワードで宣言された仮定は、デフォルトでハード厳格度になります。キーワードで宣言された仮定をソフトにするには、`#[rigidity = "soft"]` で注釈を付けてください。

パラメータをデフォルト値に固定する

例示タスクを実行する際、ソースファイルにデフォルト値が指定されているパラメータは自動的にその値に固定されます。そのため、例示器は、パラメータがそれらの指定された値をとるモデルのみを見つけることになります。

場合によっては、デフォルト値が仕様や仮定と互換性がない場合、充足不可能になることがあります。VSCode拡張機能の *easy analysis* モードでは、「Fix Params to Defaults」オプションのチェックを外すことでこの動作を無効にできます。この場合、例示器はパラメータにデフォルト値がないかのように動作します。Python SDKでは、`exemplify` メソッドに `fix_defaults = False` を渡すことで同じ効果を得られます。

特定のパラメータは自由に変動させつつ、他のパラメータをデフォルト値に固定したい場合は、例示を実行する際にそれらのパラメータに `null` というJSON値を渡すことで実現できます。

具体的な例として、次のシステムを考えてみましょう：

```
system VendingMachine

#[default = 1]
param cansMax : Int

signal cans : Int

assumption cansPositive = cans >= 0

spec reasonableCans = 0 < cans < cansMax
```

この仕様は `cansMax` が1（デフォルト値）の場合、`0 < cans < 1` を満たす整数 `cans` が存在しないため、充足不可能です。したがって、「Fix Params to Defaults」オプションがチェックされている場合、充足不可能と判定されます。しかし、オプションのチェックが外れている場合、例示器は `cansMax` に別の値（例：5）を自由に選択でき、仕様が充足可能になります。同じ効果を得るもう一つの方法は `{ cansMax: null }` を渡すことで、これにより、例示器は `cansMax` を自由変数として扱うようになります。

例

温度センサーシステムに `always_in_bounds` という仕様があるとします。混合した厳格度レベルで例示を実行することができます：

- **ハード**仮定 `"always_in_bounds"` は、生成されたトレースが境界を守ることがを保証します。これが不可能な場合、ソルバーは失敗します。

- **ソフト**仮定 "eventually (temperature > 30)" は、トレースに高温度の読み取りを含めることが望ましいという優先条件を表しますが、これがハード制約と矛盾する場合でも、ソルバーは結果を生成します。

データファイル

SpecForgeはシグナルデータとして CSV、JSON、JSONL（NDJSONとも呼ばれる）の3つのデータ形式をサポートしています。パラメータはJSONファイルで別途指定します。

シグナルデータ

各データサンプルには `time` フィールドと1つ以上のシグナル値があります。 `time` フィールドは必須で、正確に `time` という名前でなければなりません（大文字・小文字を区別）。任意の数値（整数または浮動小数点数）を受け付け、サンプルは時刻の昇順で並べる必要があります。

CSV

各列ヘッダーがシグナル名に対応します。レコード型シグナルはセパレータ（デフォルトは `_`）でフラット化されます。

```
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
```

CSV内の真偽値は `true / false`、`True / False`、または `1 / 0` で記述できます。

列挙型コンストラクタはソースファイルと同じように記述します。ただし、データファイルにはモジュールやインポートの概念がないため、コンストラクタには名前のみを含めることができ、モジュールや型による修飾はできません（例：`#Color::Red` や `#Utils::Color::Red` は許可されず、`#Red` と記述する必要があります）。データファイル内のコンストラクタは適切に曖昧性が解消されます。

JSON

`time` キーと `value` キーを持つオブジェクトのJSON配列です。レコード型シグナルにはネストしたJSONオブジェクトを使用できます:

```
[
  {
    "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 }
    }
  }
]
```

フラット化することもできます（[レコードエンコーディング](#)を参照）:

```
[
  {
    "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

JSONと同じですが、外側の配列なしで1行に1オブジェクトずつ記述します。

```
{"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}}}
```

列名

データファイルの列名はスペック内のシグナル宣言（またはそのエイリアスのいずれか）と一致する必要があります。シグナル名が列名にどのように対応するかは、コンポーネント階層とレコードエンコーディングの2つの要素によって決まります。

注意: データファイル内でシグナル宣言と一致しない余分な列は、警告なく無視されます。

コンポーネント階層

システムコンポーネントのシグナルには、`::` で区切られたコンポーネントパスがプレフィックスとして付きます。例えば、Battery システムが次のように定義されている場合:

```
system Battery

signal level: Float
signal voltage: Float
```

そして、それを使用する Vehicle が次のように定義されている場合:

```
system Vehicle

signal speed: Float

component battery: Battery
```

データファイルには speed（直接シグナル）と battery::level、battery::voltage（Battery コンポーネントのシグナル）の列が必要です。

CSVでは次のようになります:

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

JSONでは、コンポーネントをネストしたオブジェクトとして記述することもできます:

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

レコードエンコーディング

レコード型のシグナルはフィールドを別々の列として表現する必要があります。エンコーディングモードは2種類あります。

フラット (デフォルト): レコードフィールドをセパレータ (デフォルトは `_`) で連結します。これはすべての形式に適用されます。例えば、次のシグナルが定義されている場合:

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

列は `gps_lat` と `gps_lon` になります。

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

カスタムセパレータを指定することができます。セパレータには `,`、`"`、`\`、`::` を含めることはできません。

ネスト: レコードフィールドをネストしたJSONオブジェクトとして表現します。これはJSONとJSONLにのみ適用されます。

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

エイリアス

シグナルまたはパラメータに**エイリアス**が設定されている場合、データファイルの列名 (またはフィールド名) として、ソースファイルで宣言された名前の代わりにそのエイリアスを使用できます。列名 (またはフィールド名) には、曖昧さが無い限り、エイリアス (言語を問わない) と正規名を任意に組み合わせで使用できます。

例えば、以下のシステムの場合、

```
system Vehicle

#[alias(en = "Velocity", ja = "速度")]
signal speed: Float

#[alias(en = "Pressure", ja = "圧力")]
signal pressure: Float

#[alias(en = "Temperature", ja = "温度")]
signal temp: Float
```

CSVファイルでは正規名とエイリアスを組み合わせて使用できます:

```
time,Velocity,圧力,temp
0.0,30.0,101.325,295.0
```

レコード型のシグナルおよびパラメータでは、エイリアスはシグナルのルート名を置き換えます。例えば、`#[alias(en = "Position")] signal gps: { lat: Float, lon: Float }` に対しては、平坦な

記述方式ではカラム名が `Position_lat` と `Position_lon` に、階層的な記述方式ではオブジェクトキーが `Position` になります。レコードのフィールド名 (`lat`、`lon`) は影響を受けません。

マップされたシグナル

[コンポーネント](#)をインスタンス化する際、そのシグナル（またはパラメータ）を式にマップすることができます。マップされたシグナルはデータから読み取るのではなく、親システムの他のシグナルから計算されるため、データファイルに列を必要としません。

例えば、2つのシグナルを持つ `PowerUnit` が次のように定義されている場合:

```
system PowerUnit

param max_output: Float

signal output: Float
signal temperature: Float
```

`Vehicle` は `temperature` を式にマップできます:

```
system Vehicle

signal speed: Float
signal ambient_temp: Float

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

ここで `power::temperature` は親シグナル `ambient_temp` と `speed` から計算されるため、データファイルに列は不要です。同様に、`power::max_output` は `100.0` にマップされているため、パラメータファイルに含める必要はありません。マップされていない `power::output` のようなシグナルのみがデータ列を必要とします。

パラメータファイル

パラメータファイルの構造は、スペック内のパラメータ宣言を反映します。

`Vehicle` システムは独自のパラメータを宣言し、`PowerUnit` のパラメータに値を提供します:

```
system Vehicle

param temp_threshold: Float

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

対応するパラメータファイル:

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

[デフォルト値](#)を持つパラメータは省略できます。上記の例では、`power::max_output` のデフォルト値は `100.0` であるため、ファイルに含まれていません。

すべてをまとめる

上記の `Battery` と `PowerUnit` システムを組み合わせます:

```

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
}

```

パラメータファイルには temp_threshold を指定し、power::max_output（デフォルト値あり）は省略します:

```

{
  "temp_threshold": 50.0
}

```

このシステムのCSVデータファイル（フラットエンコーディング、_ セパレータ）:

```

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

```

`fuel level` はバッククォートなしで fuel level として表示され、power::temperature はマッピングされたシグナルであるため省略されています。

同じデータのJSON（ネストエンコーディング）:

```

[
  {
    "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 }
    }
  }
]

```

schema コマンド

[コンポーネント](#)の深い階層からなる複雑なシステムを扱う場合、シグナルファイルやパラメータファイルが正しい形式になっているかを確認する作業は煩雑になりがちです。 [schema コマンド](#)を使うと、期待されるシグナル／パラメータの一覧表示、データファイルの雛形生成、既存ファイルとシステムのスキーマとの照合チェックが行えます。

注意事項

- シグナル型が Float の場合、整数値は自動的に Float に変換されます。
- データ形式はファイル拡張子（.csv、.json、.jsonl）から自動検出されますが、`--file-format` CLIフラグで上書きすることもできます。

反例探索

反例探索とは、与えられた仕様に違反するシステムへの入力を発見することです。一般に、反例探索問題は以下の要素から構成されます。

- コンテキスト：システムモデル（入力信号と出力信号の関係）
- 入力：システムの入出力信号に関する仕様
- 出力：仕様違反を引き起こす入力信号

反例探索ツールは以下のように分類されます。

- **System-Dependent（システム依存型）**：特定のシステムモデルに対して使用できる falsifier。
- **System-Independent（システム非依存型）**：任意のシステムモデルで使用できる falsifier。つまり、ユーザーがモデルを選択できます。
 - **Black-box（ブラックボックス型）**：falsifier がモデルの入出力のみを通じてモデルを理解するシステム非依存型の falsifier。
 - **Glass-box（ガラスボックス型）**：falsifier がモデルの実行ハンドルではなくモデルの記述を受け取るシステム非依存型の falsifier。

SpecForge は現在、*System-Dependent* 型の falsifier とのインターフェースのみをサポートしています。つまり、ユーザーはモデルを固定し、そのモデルに特化した falsifier を提供するスクリプトを用意する必要があります。

以下では、このような falsifier を SpecForge と連携させるためのセットアップ方法の詳細を説明します。完全な例については、[SpecForge releases](#) ページに含まれている 反例探索のサンプルプロジェクトを参照してください。

- **F16 航空機モデル**：16 個の連続変数と区分的非線形微分方程式を用いてモデル化された [F16 航空機](#) のモデルで、中核部分は地面衝突回避システムです。
- **オートマチックトランスミッション**（MATLABが必要）：2つの入力（スロットルとブレーキ）を持つ車両のギア選択モデルです。モデルの他の要素には速度や回転数が含まれます。このモデルは [MATLAB ドキュメントの例](#) に一般的に含まれるものにインスパイアされており、Simulink を用いて実装されています。

反例探索スクリプトの準備

反例探索スクリプトは以下の引数を受け取れる必要があります。

- `--project-dir` : SpecForge プロジェクトディレクトリのパス（`specforge.toml` ファイルが置かれている場所）
- `--system` : falsify するシステムの名前
- `--spec` : falsify する仕様の名前
- `--params` : JSON 形式のシステムの `param` の値
- `--options` : スクリプトへの追加オプション

したがって、コマンドは以下のように呼び出されます。

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

反例探索スクリプトの出力は以下のいずれかである必要があります。

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

ここで、`err` は文字列（例：「Couldn't Falsify」）であり、`signal` は JSON 形式の 反例探索の結果です。pandas データフレームを使用している場合、これは SpecForge SDK の `converters.python_to_api_timeseries` 関数を使って取得できます。

Falsifier の登録

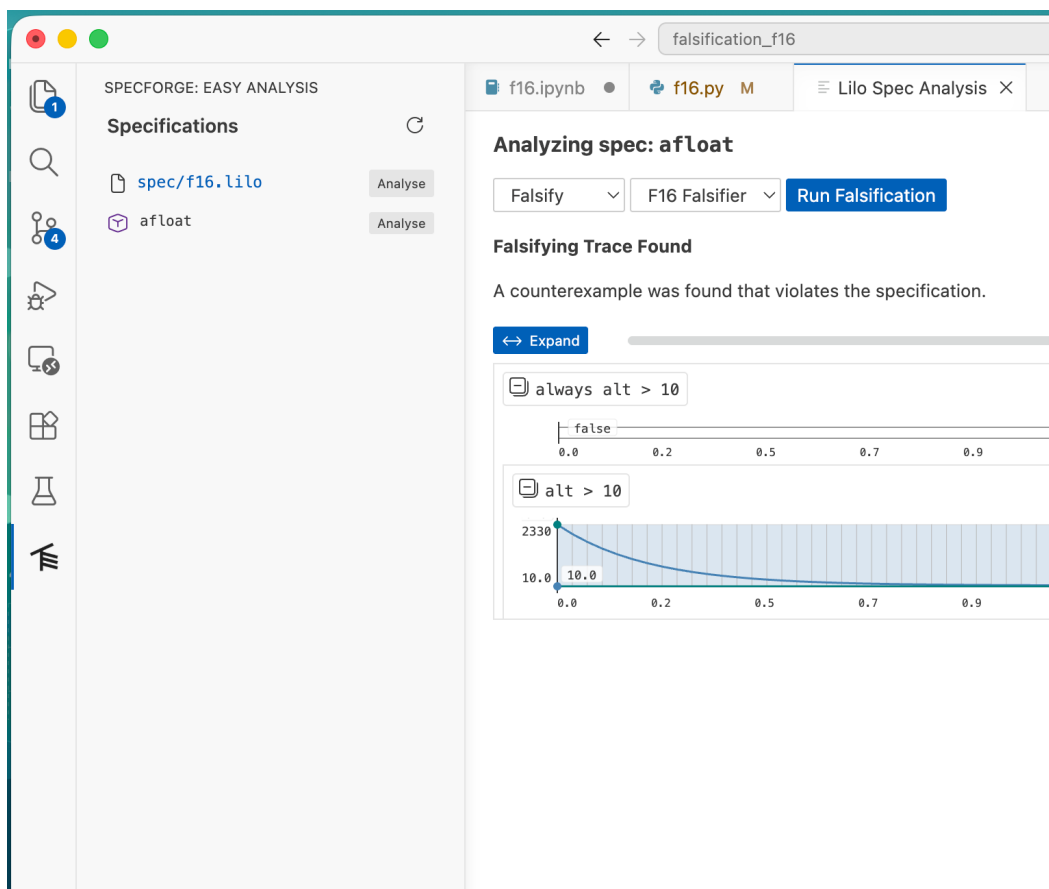
specforge.toml ファイルに、falsifier を以下のように登録します。

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

これにより、システム automatic_transmission に関連付けられた falsifier がスクリプト scripts/automatic_transmission.sh の実行により呼び出せることを SpecForge に伝えます。

反例探索ワークフローの実行

反例探索スクリプトの準備と登録が完了したら、VSCode 拡張機能の *easy analysis* オプションを使って実行できます。以下のスクリーンショットを参照してください。



オンラインモニタリング

SpecForgeには、Liloで記述された仕様をオンラインモニタリングする機能があります。オフラインモニタリングがトレース全体を処理した後一括して判定を出すのとは対照的に、トレースを取り込みながらストリーミング方式で判定を出すことを意味します。

オンラインモニタを起動するには、specforge CLIの streammon コマンドを使用してください。必要な引数については specforge streammon --help を実行してください。specforge streammon コマンドを実行するとモニタが起動し、STDINからトレースを取り込み、STDOUTに出力信号を生成します。

モニタはCSVまたはJSONL形式のトレースを受け付けます。CSVの場合、最初の行はヘッダーでなければならず、そのヘッダーは各信号に対応する列を識別するために使用されます。JSONLの場合、各行が信号名をキーとするJSONオブジェクトでなければなりません。RECORD_ENCODING サブコマンドを使用して、レコードが flat または nested 形式でエンコードされているかどうかを指定できます ([データに関する章](#)を参照)。

オンラインモニタの現在の実装は、時相演算子の区間が将来または過去の小さなウィンドウに限られている場合に最も良好に動作します。非有界な長さを持つ区間は、たとえ過去の区間であってもサポートされていません。

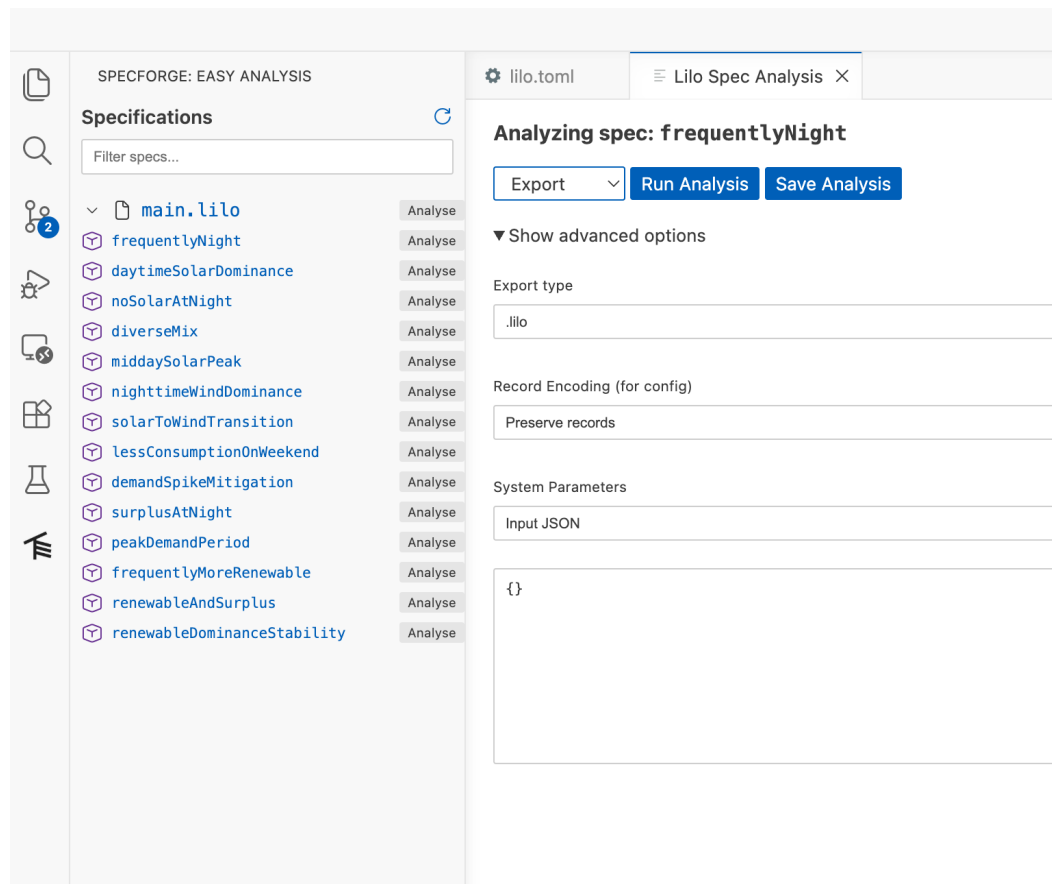
将来時相の時相演算子はオンラインモニタでサポートされています。それらの判定は、仕様内の最大の将来ウィンドウの分だけ遅延して生成されます。例えば、式 eventually [0, 5] f をモニタリングする場合、モニタは現在時刻 t の判定を、時刻 t + 5 までのトレースを取り込んでから生成します。

RTAMTへのエクスポート

SpecForgeには、Liloで記述された仕様をサードパーティのモニタリングツールで使用できるようにエクスポートする機能があります。現在は、Nickovic氏らによって開発・維持されているSignal Temporal Logic (STL) のモニタリングツール**RTAMT**のみをサポートしています。

エクスポート機能の呼び出し

エクスポート機能は、以下のスクリーンショットに示されているように、SpecForge VSCode拡張機能のeasy analysis UIから呼び出せます。



また、PythonスクリプトやJupyter NotebookでPython SDKを使用して呼び出すこともできます。この方法の詳細については、[Python SDKドキュメント](#)を参照してください。

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

RTAMTに関する補足事項

RTAMTはPythonライブラリとして配布されています。インストールと使用方法については、[RTAMTドキュメント](#)を参照してください。

RTAMTにエクスポートする際、SpecForgeは定義をインライン展開して単一の式を生成します。パラメータはデフォルト値、またはエクスポートの引数として指定された値で置き換えられます。値が提供され

ていないパラメータは、同じ名前の自由変数で置き換えられます。パラメータにデフォルト値がある場合でも自由変数で置き換えたい場合は、パラメータ値をJSON値 `null` に設定してください。

RTAMTの体系はLiloよりも単純であるため、LiloのRTAMTへのエクスポートが一部のケースでできない場合があります。RTAMTでサポートされていないLiloの機能には、文字列やレコード、`time` 変数、区間内の非自明な式、`if`、`did_change`、`will_change` などがあります。SpecForgeが適用する最適化によってサポートされていない機能が除去される場合は、エクスポートが成功します。

スライディングウィンドウの量的演算子 `max_future`、`min_future`、`max_past`、`min_past` は、RTAMTでは対応する時相演算子（`eventually`、`past`、`always`、`historically`）に変換されます。これはRTAMTのロバストネスモードでは、式の型検査を行わないため機能します。

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、例示、反例探索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。

サーバー接続

- **API Base URL** (`specforge.apiBaseUrl`)：バックエンド解析サーバーのベースURL。
`spawnServer` が有効な場合は無視されます。(文字列、デフォルト：`http://localhost:8080`)
- **Spawn Server** (`specforge.spawnServer`)：有効にすると、拡張機能が `apiBaseUrl` の外部サーバーに接続する代わりに、SpecForgeサーバープロセスをローカルで管理します。(ブール値、デフォルト：`false`)
- **Server Path** (`specforge.serverPath`)：SpecForgeバイナリのコマンドまたは絶対パス。
`spawnServer` を有効にする必要があります。(文字列、デフォルト：`specforge`)
- **Preferred Port** (`specforge.preferredPort`)：ローカルで起動するSpecForgeサーバーの優先ポート。拡張機能はこのポートの使用を試み、使用中の場合は別の利用可能なポートにフォールバックします。`spawnServer` を有効にする必要があります。(数値、デフォルト：`8080`)

サーバーチューニング

これらの設定は `spawnServer` が有効な場合にのみ適用されます。

- **Cache Bound** (`specforge.cacheBound`)：SMTソルバー結果のLRUキャッシュのサイズ。デフォルト値を使用する場合は空白のままにします。(文字列、デフォルト：`""`)
- **Semaphore Limit** (`specforge.semaphoreLimit`)：同時に実行できるSMTソルバープロセスの最大数。デフォルト値を使用する場合は空白のままにします。(文字列、デフォルト：`""`)

LLM統合

これらの設定は、仕様生成と診断説明に使用されるAI機能を構成します。プロバイダーとモデルの設定は `spawnServer` が有効な場合にのみ適用されます。

- **LLM Provider** (`specforge.llmProvider`) : 説明と仕様生成に使用するLLMプロバイダー。オプション: `openai`、`anthropic`、`ollama`、`gemini`、`default`。(文字列、デフォルト: `default`)
- **LLM Model** (`specforge.llmModel`) : 説明と仕様生成に使用するLLMモデル。デフォルト値を使用する場合は空白のままにします。(文字列、デフォルト: `""`)
- **API Key Source** (`specforge.apiKeySource`) : LLMプロバイダーのAPIキーをどのように取得するかを指定します。これを `env` に設定すると、APIキーは環境変数から継承されます。 `vscode` に設定すると、APIキーはVS Code UIを通じて入力され、VS CodeのSecretStorageに安全に保存されます。(文字列、デフォルト: `vscode`)
- **Ollama API Base** (`specforge.ollamaApiBase`) : OllamaのAPIベースURL。環境から継承する場合、または関係ない場合は空白のままにします。(文字列、デフォルト: `""`)

APIキーの構成

`specforge.apiKeySource` が `vscode` (デフォルト) に設定されている場合、APIキーはVS CodeのSecretStorageに保存されます。コマンドパレット (`Ctrl+Shift+P` / `Cmd+Shift+P`) から次のコマンドを使用して追加または削除できます。

- **SpecForge: Configure API Key for LLM Provider** : 現在選択されているプロバイダー (`specforge.llmProvider`) のAPIキーを入力する箇所です。空のキーを入力すると、そのプロバイダーの保存されたキーが削除されます。
- **SpecForge: Clear Stored API Keys for LLM Providers** : SecretStorageに保存されているすべてのAPIキーを削除します。

設定されたプロバイダーにAPIキーが必要であるにもかかわらず見つからない場合、キーの構成や設定を開くためのショートカットを含む警告が表示されます。この状態でも、LLM機能なしでSpecForgeを使用すること自体は可能です。

`specforge.apiKeySource` が `env` に設定されている場合、起動されたサーバーは環境変数 (`OPENAI_API_KEY`、`ANTHROPIC_API_KEY`、`GEMINI_API_KEY`) からAPIキーを継承します。

その他の設定

- **Enable Preview Features** (`specforge.enablePreviewFeatures`) : まだリリース準備ができていない実験的機能を有効化します。(ブール値、デフォルト: `false`)
- **Trace Server** (`specforge.trace.server`) : VS Codeと言語サーバー間の通信をトレースします。拡張機能の問題のデバッグに便利です。オプション: `off`、`messages`、`verbose`。(文字列、デフォルト: `off`)

新しいプロジェクトの初期化

VSCodeから[推奨されるプロジェクト構造](#)でディレクトリを初期化できます。コマンドパレット (`Ctrl+Shift+P` / `Cmd+Shift+P`) を開き、**SpecForge: Initialize SpecForge Project**を実行します。これは `init CLI`コマンドの実行と同等です。

このコマンドはSpecForgeバイナリに依存するため、`specforge.spawnServer` が有効で、`specforge.serverPath` が正しく構成されている場合にのみ動作します。

VSCodeでSpecForgeプロジェクトに取り組む際は、プロジェクトルートディレクトリを唯一のワークスペースフォルダとして開くことをお勧めします。

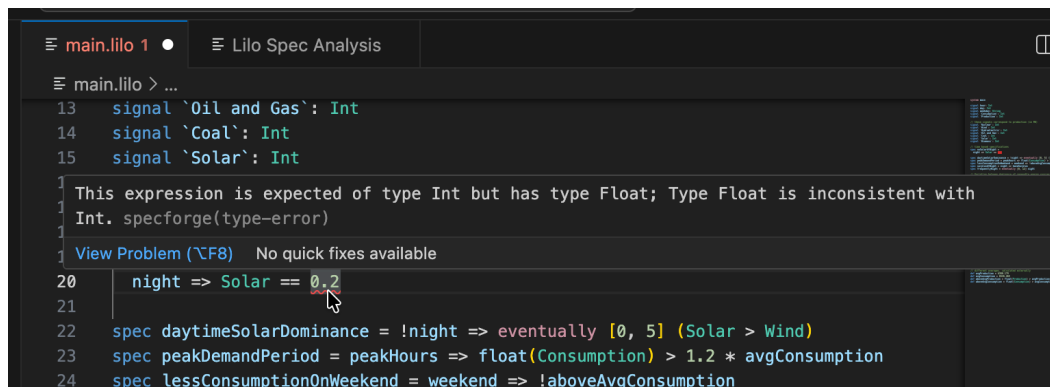
エディションとライセンス

Liloファイルを開いている間、ステータスバーに現在のエディションが表示されます。クリックすると利用水準を変更できます。詳細については、[エディション](#)、[利用水準](#)、[ライセンス](#)を参照してください。

言語機能

解析と型検査

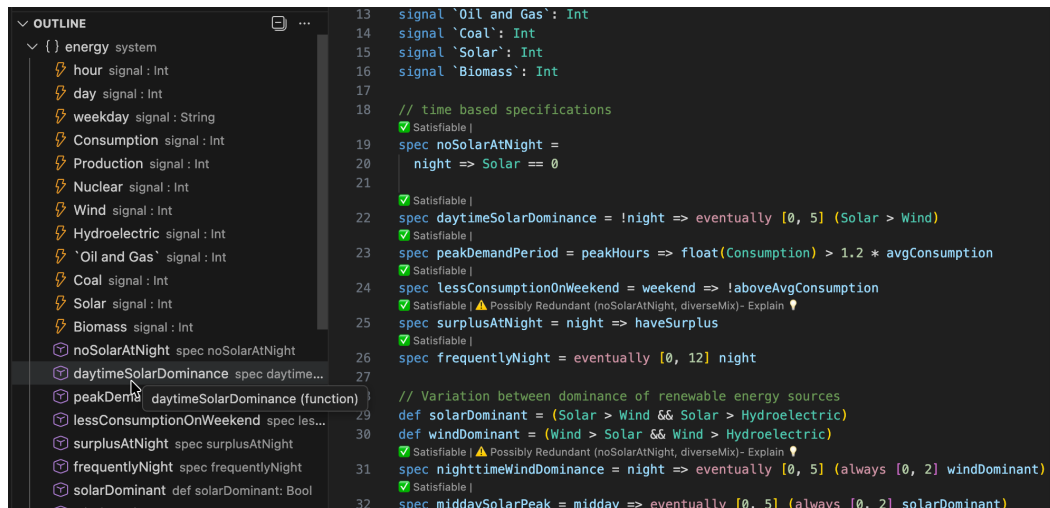
拡張機能は、仕様を記述する際にリアルタイムでチェックを実行します。エラーには下線が引かれます。該当箇所のコードにカーソルを合わせると、エラーが表示されます：



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。

```

15 signal `Solar`: Int
16 signal `Biomass`: Int
17
18 signal night_time: Bool
19
20 /// There no solar energy use during the night.
21 ✓ Satisfiable |
22 spec no_solar_at_night =
23   night_time == true => Solar == 0
24
25

```

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。

```

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

```

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

```

仕様が充足可能な場合、「✓ Satisfiable」レンズはクリックできます。クリックすると、その仕様に対して例示解析があらかじめ選択された状態でSpec Analysisペインが開き、仕様の充足可能性を示すトレース例をすぐに生成できます。

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：

```

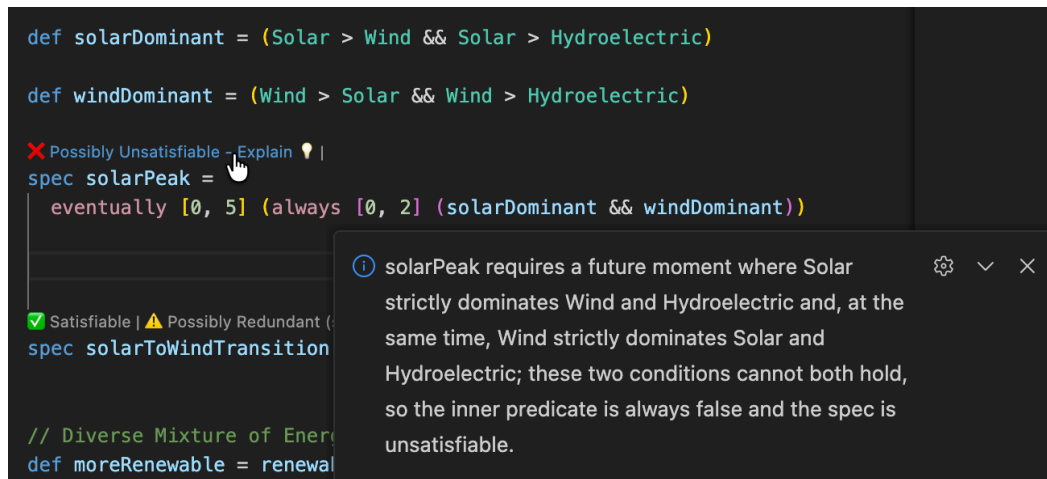
def solarDominant = (Solar > Wind && Solar > Hydroelectric)

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

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

```

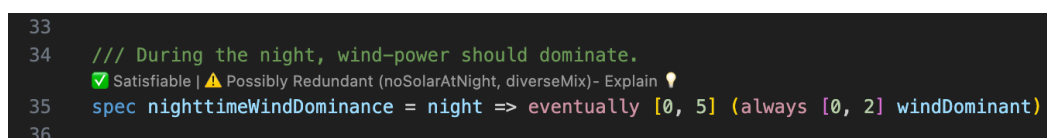
ユーザーは説明を求めることができます：



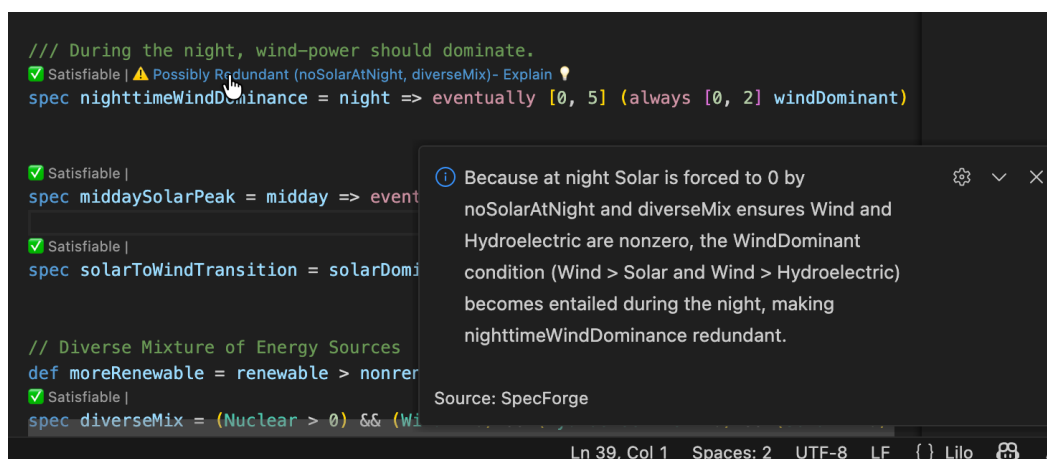
SpecForgeが充足可能性を判断できなかった場合、タイムアウトを長くして解析を再試行することができます。

冗長性

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

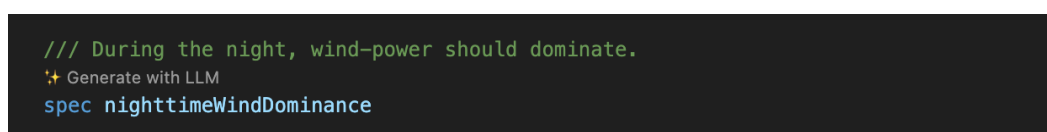


この仕様は `noSolarAtNight` と `diverseMix` によって暗黙的に必ず成り立つため、書く必要がないということをSpecForgeが示しています。Explain をクリックすると、冗長性の説明が生成されます：



スタブ

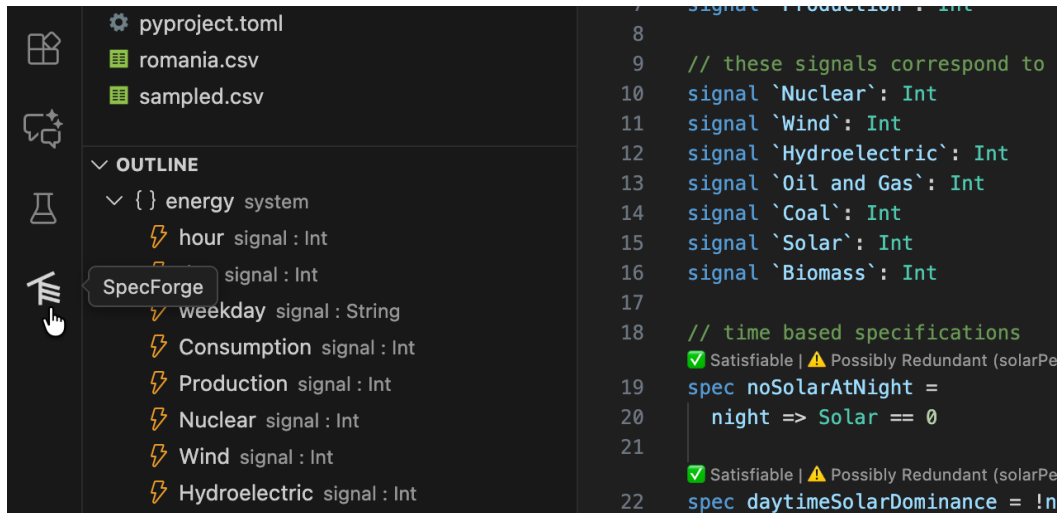
実体をもたない spec および assumption はスタブと呼ばれます。この場合、コードレンズはAIを使用した仕様生成を提案します。



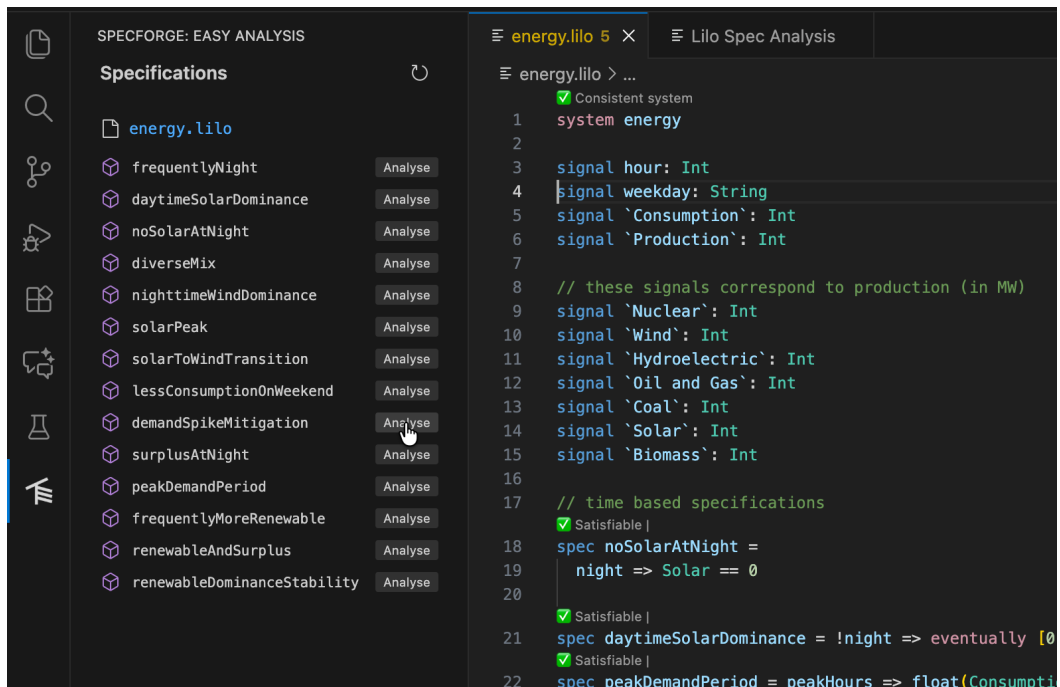
Generate with LLM をクリックすると、現在のシステムで機能する仕様の定義が生成されます。仕様があまりにも曖昧である場合、または生成に他の障害がある場合は、エラーメッセージが表示されます。

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：

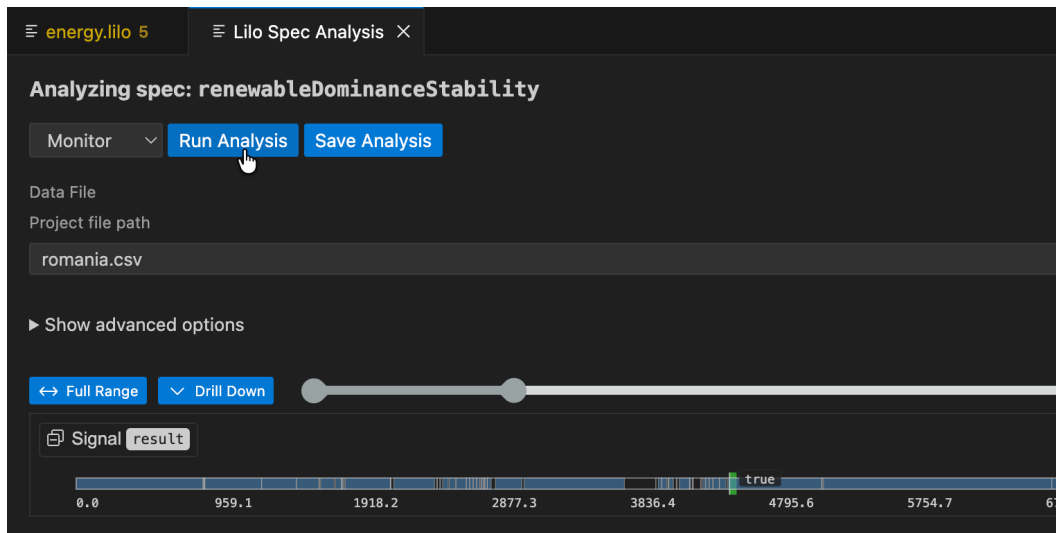


サイドバーには、すべての仕様ファイルと定義されている仕様がリストされます：

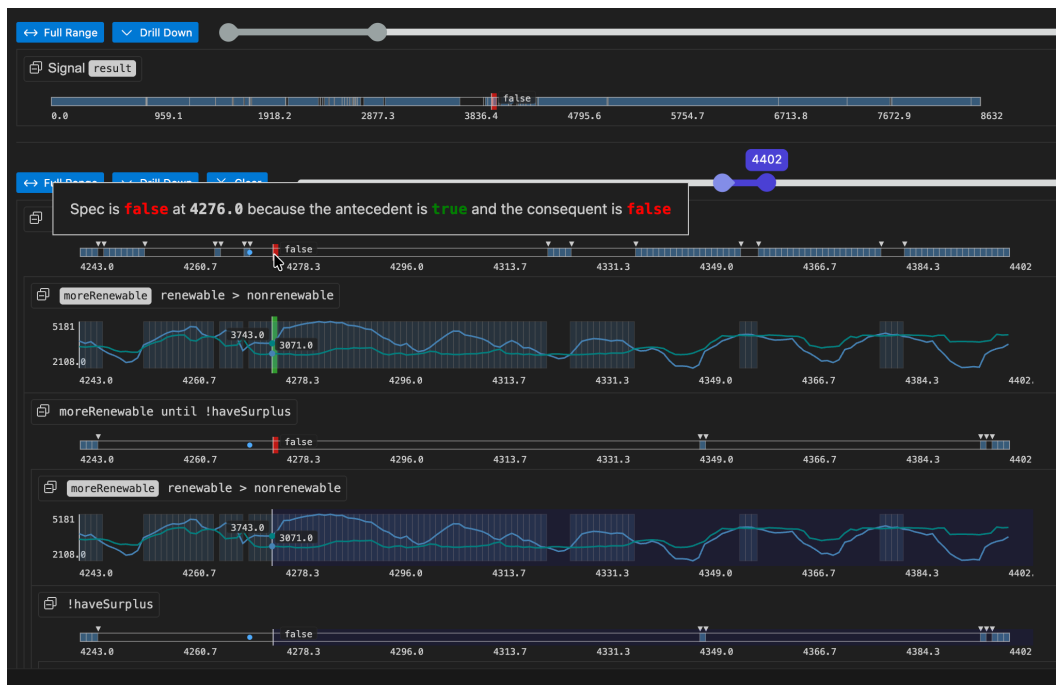


仕様の横にある Analyze をクリックすると、仕様解析ウィンドウが表示されます。これを使用して、モニタリング・例示・エクスポート・アニメーション・反例探索といったさまざまな仕様解析タスクを起動できます。スタブ（すなわち、実体をもたない仕様や仮定）の場合は、代わりに Generate ボタンが表示され、Generate with LLM のコードレンズと同様の機能を果たします。

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。

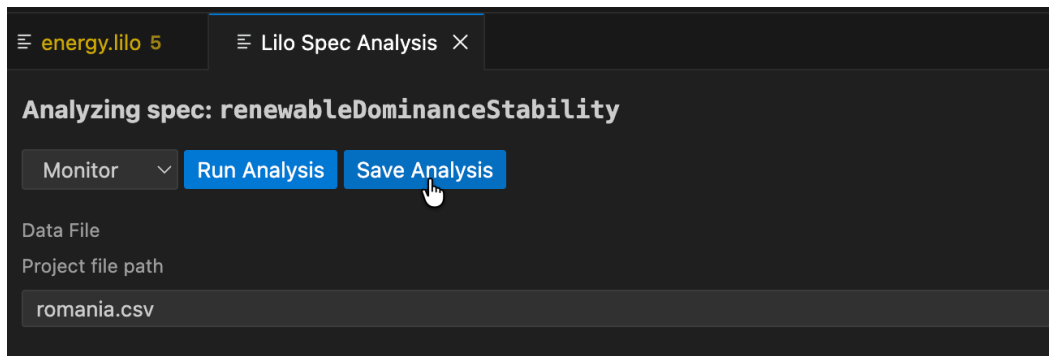


解析の結果が表示されます。青い領域は、仕様が真である時間を表します。大きなデータファイルの場合、ブール値の結果のみが表示されます。仕様がある時点で偽である理由をよりよく理解するには、ポイントを選択して Drill Down をクリックします。



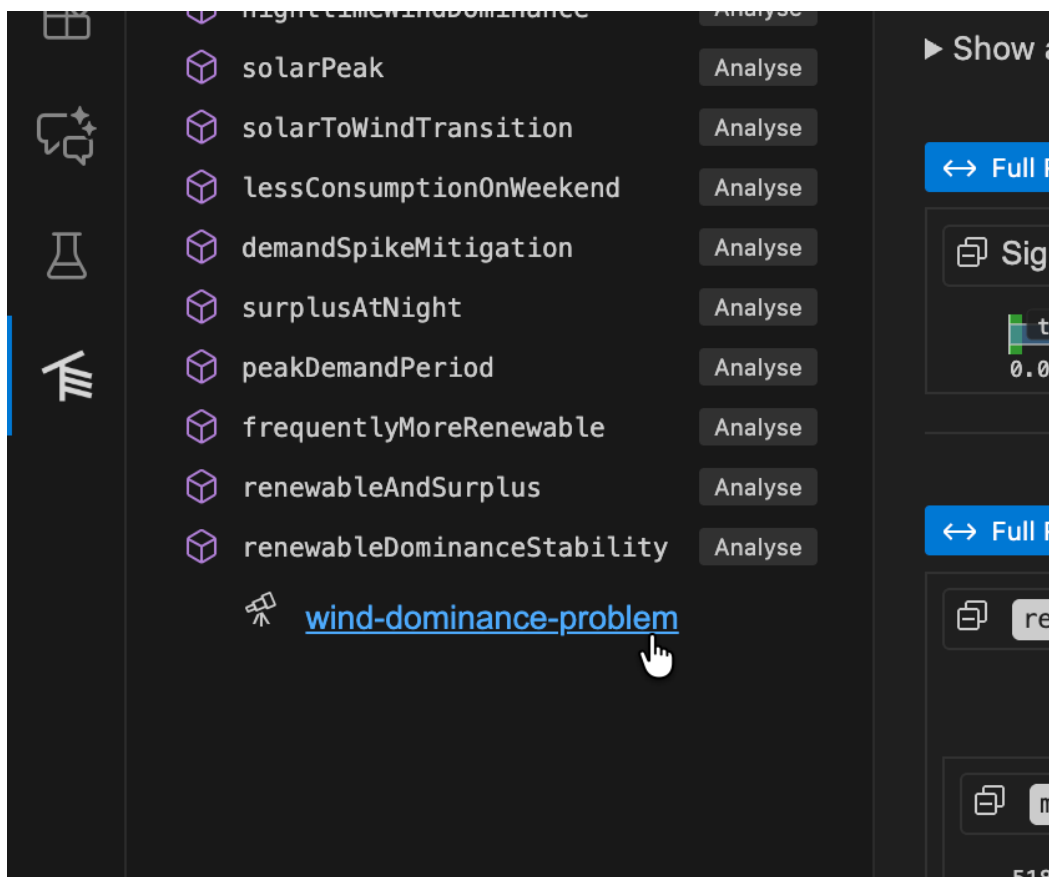
ドリルダウンは、デバッグツリーを表示するために、完全なデータソースの十分に小さいセグメントを選択しました。これは、仕様全体のモニタリング結果のツリーを表示します。各ノードは折りたたみまたは展開できます。タイムライン上にカーソルを合わせると、サブ式の結果タイムラインで関連する領域も強調表示されます。タイムライン上にカーソルを合わせると、そのサブ式のその時点での結果が真または偽である理由の説明が表示されます。

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**：ソルバーが満たすべき追加の制約で、それぞれに厳格度レベル（HardまたはSoft）が付きます。仮定と厳格度レベルの詳細については、[例示と充足可能性](#)を参照してください。

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：

- `specforge.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**：ブ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
- **System Parameters**（オプション）：システムのパラメータ値。JSONファイルまたはインライン値として提供します。Liloシステム定義内の `param` 宣言に値を供給します。

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：

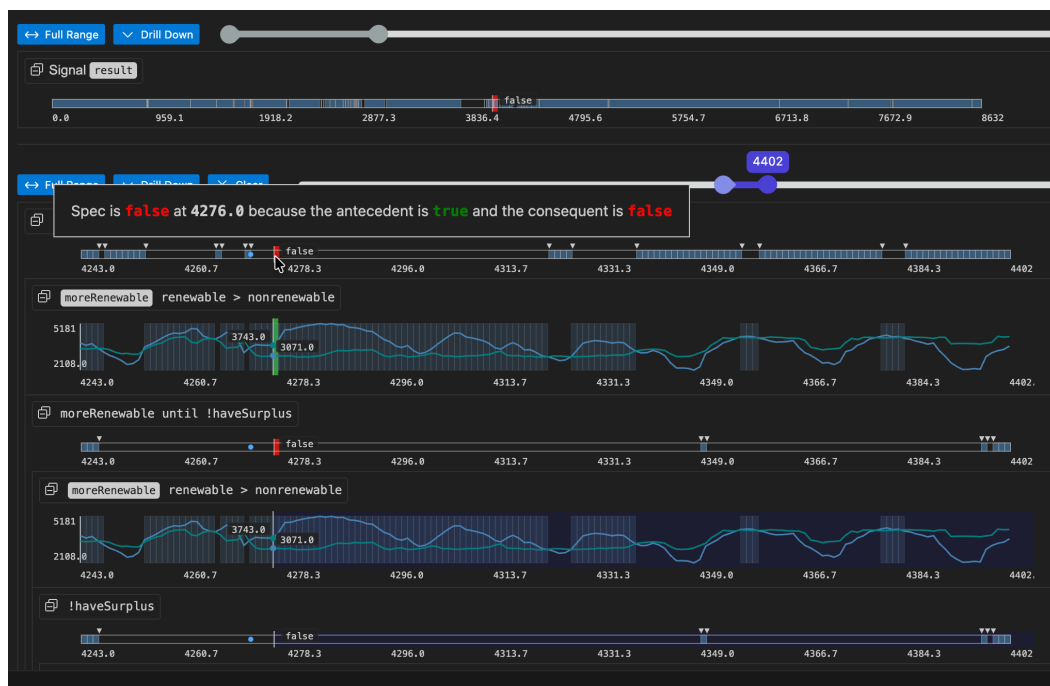
- **Time Bounds**：モニタリングを特定の時間範囲に制限
- **Sampling**：時間解像度を調整
- **Signal Filtering**：特定のシグナルのみをモニタリング
- （追加のオプションが利用可能な場合があります）

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。

仕様のフィルタリングとソート

仕様ステータススペースのフィルターボックスを使用して、表示される仕様を絞り込むことができます。検索クエリは、仕様名、docstring、エイリアスに対するテキスト検索と、[カスタムフィールド](#)に対する条件の組み合わせにすることができます。例えば、クエリは `priority:>10 reviewed:true owner:"ops"` のようにすることができます。

クエリ言語の仕様については、[仕様検索](#)のドキュメントを参照してください。

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：

- specforge.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
- **充足可能性**: 仕様、インライン式、またはシステム全体が充足可能かどうかをチェック
- **妥当性**: システムの仮定のもとで仕様または式が証明可能かどうかをチェック
- **同値性**: 2つの仕様または式が論理的に同値かどうかをチェック
- **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 `specforge.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 | None): モニタリングする仕様の名前。None の場合、システム内の**すべての仕様**を一度にモニタリングします (システム全体のモニタリング)
- `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)
- `partial_data` (bool): データがシステムで宣言されたすべての信号を含まない場合でも、モニタリングを許可します (デフォルト: True)。有効にすると、データにない信号はエラーの原因とはならず、不明として扱われます。すべての信号の値をデータに含めることを必須にするには `False` に設定します
- `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)

`definition=None` の場合、モニターはシステム内の**すべての仕様**を提供されたデータに対して評価し、結果をまとめて返します。システム全体でどの仕様が合格し、どの仕様が不合格かをすばやく把握するのに役立ちます。

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
- params_file (str/Path): システムパラメータの値を含むJSON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
- 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
)

# システムパラメータを指定してアニメーションを作成
specforge.animate(
    system="temperature_sensor",
    svg_file="temp.svg",
    data_file="sensor_data.csv",
    params={"temp_threshold": 25.0}
)

# ファイルからパラメータを読み込んでアニメーションを作成
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, ...)

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）
 - ライブラリで定義されている EXPORT_LILO、EXPORT_JSON、EXPORT_RTAMT のいずれかを 使用します
- 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
)

```

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

単一の仕様、インライン式、またはシステム全体の充足可能性を検査します。

主なパラメータ:

- **system** (str): 定義を含むシステム
- **definition** (str | None): 検査する仕様の名前
- **expression** (str | None): 指定されたシステムのコンテキストで検査するインラインLiloブール式
 - **注意:** definition または expression のうち最大1つを指定してください
- **definition** と **expression** が両方とも None の場合、SDKはシステム全体を検査します
- **timeout** (int | float): 充足可能性検査のタイムアウト秒数 (デフォルト: 30)
- **return_details** (bool): True の場合、ブール値の代わりに構造化された充足可能性検査のレスポンスを返します (デフォルト: False)

戻り値:

- **bool**: 充足可能な場合は True、そうでない場合は False
- **return_details=True** の場合、次の2つのキーを持つ辞書を返します:
 - **"status"**: 充足可能性検査の結果。result["status"]["tag"]を確認すると、次のいずれかの値になります:
 - **"Consistent"** - 仕様は充足可能です
 - **"InConsistent"** - 仕様は充足不能です。"contents" フィールドには、充足不能コアの詳細を示す UnsatInfo オブジェクトが含まれます
 - **"Unknown"** - ソルバーは充足可能性を判定できませんでした
 - **"TimedOut"** - ソルバーがタイムアウトしました。"contents" フィールドにはマイクロ秒単位のタイムアウト時間が含まれます
 - **"param_situation"**: 検査中にシステムパラメータ (param 宣言) がどのように扱われたかを示します:
 - **"UsingDefaults"** - ソルバーはすべてのパラメータを宣言されたデフォルト値に固定し、その値で結果を得ました。最初に試行される方法です

- "UnfixedParams" – ソルバーはすべてのパラメータを自由変数（制約なし）として扱いました。デフォルト値を使用した検査で "Consistent" という結果が得られなかった場合に、自由変数として再検査します

status と param_situation の意味のある組み合わせは次のとおりです：

- ("Consistent", "UsingDefaults") – パラメータを宣言されたデフォルト値に固定すると、仕様は充足可能です
- ("Consistent", "UnfixedParams") – 仕様は充足可能ですが、パラメータを宣言されたデフォルト値に固定した場合は充足可能ではありません
- ("InConsistent", "UnfixedParams") – パラメータのどのような値に対しても、仕様は充足不能です

例:

```
# 単一の仕様を検査
is_satisfiable = specforge.check_satisfiability(
    system="temperature_sensor",
    definition="always_in_bounds"
)
print(is_satisfiable)

# パラメータの扱いを確認するために詳細なレスポンスを取得
details = specforge.check_satisfiability(
    system="temperature_sensor",
    definition="always_in_bounds",
    return_details=True
)
print(details["status"]["tag"])          # 例: "Consistent"
print(details["param_situation"])        # 例: "UsingDefaults"または"UnfixedParams"

# システム全体を検査してブール値を取得
system_is_satisfiable = specforge.check_satisfiability(
    system="temperature_sensor"
)
print(system_is_satisfiable)

# システムのコンテキストでインライン式を検査
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, ...)

システムの仮定のもとで、単一の仕様またはインライン式の妥当性を検査します。

主なパラメータ:

- system (str): 定義を含むシステム、または式が解析されるコンテキストとなるシステム
- definition (str | None): 証明する仕様の名前
- expression (str | None): 指定されたシステムのコンテキストで証明するインラインLiloブール式
 - **注意:** definition または expression のいずれか1つのみを指定してください
- timeout (int | float): 妥当性検査のタイムアウト秒数（デフォルト: 30）
- return_details (bool): True の場合、ブール値の代わりに構造化された妥当性検査のレスポンスを返します（デフォルト: False）

戻り値:

- bool: 対象が妥当な場合は True、そうでない場合は False
- return_details=True の場合、SpecForge APIから返された構造化された妥当性検査の結果を返します。結果については result["status"]["tag"] を、デフォルト値を持つパラメータの扱いについては result["param_situation"] を確認してください。

例:

```

# 単一の仕様を検査
is_valid = specforge.check_validity(
    system="temperature_sensor",
    definition="always_in_bounds"
)
print(is_valid)

# システムのコンテキストでインライン式を検査
expr_is_valid = specforge.check_validity(
    system="temperature_sensor",
    expression="always (min_temperature <= temperature <= max_temperature)"
)
print(expr_is_valid)

# 必要に応じて詳細なレスポンスを取得
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, ...)

システムの仮定のもとで、2つの仕様またはインライン式が論理的に同値かどうかを検査します。

主なパラメータ:

- system (str): 両辺が解析されるコンテキストとなるシステム
- left (str): 1つ目の仕様名またはインラインLilo式
- right (str): 2つ目の仕様名またはインラインLilo式
- timeout (int | float): タイムアウト秒数 (デフォルト: 30)
- return_details (bool): True の場合、ブール値の代わりに構造化された同値性検査のレスポンスを返します (デフォルト: False)

戻り値:

- bool: 2つの対象が同値の場合は True、そうでない場合は False
- return_details=True の場合、SpecForge APIから返された構造化された同値性検査の結果を返します。結果については result["status"]["tag"] を、デフォルト値を持つパラメータの扱いについては result["param_situation"] を確認してください。

例:

```

# 2つの表現が一致することを確認
are_equivalent = specforge.check_equivalence(
    system="temperature_sensor",
    left="always_in_bounds",
    right="always (min_temperature <= temperature <= max_temperature)"
)
print(are_equivalent)

# インライン式を組み合わせて検査
are_equivalent = specforge.check_equivalence(
    system="temperature_sensor",
    left="x > 0 && y > 0",
    right="y > 0 && x > 0"
)
print(are_equivalent)

# 詳細なレスポンスを取得
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, ...)

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: []）。各仮定は次の要素を持つ辞書です:
 - "expression" (str): 生成時に仮定するLiloブール式（または既存のspec/defの名前）
 - "rigidity" (str): "Hard" または "Soft"。[厳格度レベル](#)を参照してください
- 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: True）
- at_beginning (bool): True（デフォルト）の場合、例示器は**最初の**時点ですでに仕様を満たすトレースを探索します。Falseの場合、生成されたトレースの後の時点で仕様が満たされてもかまいません。生成例が最初から仕様を満たすことを示したい場合は True が便利です。False にするとソルバーの自由度が増し、トレースの後の時点で自然に満たされる past などの時相演算子を仕様を含む場合に役立ちます
- fix_defaults (bool): デフォルトは True です。True の場合、例示器はデフォルト値を持つパラメータをその値に固定します。False の場合、例示器はデフォルト値を無視し、パラメータを自由変数として扱います。詳細は[パラメータをデフォルト値に固定する](#)を参照してください
- 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）

厳格度レベル

各仮定には厳格度レベル（"Hard" または "Soft"）があり、ソルバーがその仮定を絶対に違反できない制約として扱うか、選好として扱うかを制御します。詳しくは[例示と充足可能性](#)を参照してください。

例:

```

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

# ハード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
)

# ハード仮定とソフト仮定を組み合わせる
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
)

# 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
)

# トレースの後の時点で仕様を満たすことを許可（最初の時点で満たす必要はありません）
specforge.exemplify(
  system="temperature_sensor",
  definition="recovery_spec",
  n_points=20,
  at_beginning=False
)

# 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
)

```

list_defs(system, specs_only=False)

指定されたLiloシステム内のすべての定義を一覧表示します。

パラメータ:

- system (str): 定義を一覧表示するLiloシステム
- specs_only (bool): Trueの場合、他の定義を除き仕様のみを一覧表示します（デフォルト: False）

戻り値: str の list - 定義名の一覧

例:

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

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

仕様検索クエリ

SpecForgeは仕様を整理するためのツールとして設計されており、各仕様に[docstring](#)、[ラベル](#)、[エイリアス](#)、[カスタムフィールド](#)で注釈を付けることができます。SpecForgeシステムでは、[CLI](#)または[VSCode拡張機能](#)のサイドバーを使用して、関連する仕様を問い合わせることができます。

クエリの構文と意味論

仕様検索は、[GitHub検索構文](#)や[LIQE](#)に類似したクエリ言語を使用します。

docstringや定義名に基づくあいまい検索または完全一致検索、数値フィールドの比較、ラベルによるフィルタリング、および複数のフィルターのブール結合をサポートしています。仕様は数値フィールドを使用してソートすることもできます。

基本構文

あいまい検索と完全一致のテキスト検索

テキストを使用して仕様を検索するには、クエリを直接入力します。例えば、`pump` という入力に対しては、名前、エイリアス、またはdocstringに `pump` を含む仕様を検索します。

デフォルトでは、テキスト検索はあいまい検索です。完全一致検索を行うには、引用符を使用します（例：`"pump"`）。

複数のあいまい検索語または完全一致検索語（スペースで区切られたもの）がある場合、クエリはこれらのクエリの論理和として解釈されます。例えば、クエリ `tube "pump"` は、`tube` にあいまい一致するか、`"pump"` に完全一致する仕様に一致します。

フィールドに対する述語

クエリでは、カスタムフィールドの値に基づくフィルターを指定できます。

- **比較**： `priority` という名前のフィールドがある場合、`priority:>50`、`priority:>=50` などのクエリを書くことができます。
- **真偽値**： Boolean型のフィールド `reviewed` がある場合、`reviewed:true` または `reviewed:false` と書くことができます。
- **文字列**： 文字列型のフィールド `reviewer` を参照するには、`reviewer:Sam` または `reviewer:"Samwell Tarly"` と書くことができます。
- **コンストラクタ**： 述語を使用してコンストラクタと照合することができます。これを行うには、通常通りコンストラクタの前に `#` を付けます。例：`color:#Red`

ラベル

クエリでは、ラベルを使用してフィルタリングすることが可能です。例えば、`label:todo` は `todo` ラベルを持つすべての項目に一致します。

システム

クエリでは、仕様が属するシステムに基づいてフィルタリングすることもできます。例えば、`system:engine` は `engine` システム内の仕様にのみ一致します。`system:cooling::pump` のような修飾名を指定することもできます。

依存関係

`depends-on`: 構文を使用すると、仕様の定義に使用されている他の仕様、`def`、`signal`、`param`に基づいて仕様をフィルタリングできます。例えば、`depends-on:pressure` は `pressure` という `signal`（または `pressure` という名前の `def` や `param`）に依存する仕様に一致します。`depends-on:chamber2::pressure` のような修飾名を指定することもできます。

論理結合

クエリ言語は、`AND`、`OR`、`NOT` を使った複数のクエリの論理結合による組み合わせもサポートしています。クエリをグループ化するには、括弧を使用します。

`NOT` 演算子は、`-` 接頭辞を使用して書くこともできます。例えば、`-label:todo` は `NOT label:todo` と等価です。

明示的な接続詞が使われていない場合、クエリは以下のように解釈されます：基本構文のグループがスペースで区切られている場合、あいまいテキスト検索語と完全一致テキスト検索語は論理和を用いて結合され、述語は論理積を用いて結合されます。例えば、`tube "pump" priority:>50 label:todo` は、`(tube) OR ("pump") AND (priority > 50) AND (label:todo)` として解釈されます。このほか、`(NOT priority:>10) reviewed:true label:important foo bar` は、`(NOT priority > 10) AND reviewed:true AND label:important AND (foo OR bar)` として解釈されます。

並べ替え

クエリの結果を並べ替えるには、`sort`: キーワードの後にフィールド名を指定します。例えば、クエリの末尾に `sort:priority` を追加すると、結果が `priority` フィールドの降順で並べ替えられます。

昇順で並べ替えるには、フィールド名の末尾に `-asc` を追加します。例えば、`sort:priority-asc` は、結果を `priority` フィールドの昇順で並べ替えます。同様に、`-desc` を追加すると降順で並べ替えられます。デフォルトでは、並べ替えは降順です。

複数の `sort` クエリを使って、同点時の並べ替えルールを指定することができます。例えば、`sort:priority sort:created_at` は、結果を `priority` の降順で並べ替え、同点の場合は `created_at` の降順で並べ替えます。

並べ替えは、スペースで区切ることでフィルタリングクエリと組み合わせることができます。例えば、`label:todo sort:priority` は、`todo` ラベルを持つ仕様をフィルタリングし、`priority` フィールドで並べ替えます。

検証

クエリの結果が論理的な整合性条件を満たしているかを確認したい場合がしばしばあるかと思います。例えば、`todo` ラベルを持つすべての仕様の `priority` フィールドが50より大きいかを確認したい場合などです。

`validate` [CLIコマンド](#)を使うと、前件クエリの結果が後件クエリの結果に含まれていることを確認できます。上記の例の場合、ターミナルで以下のコマンドを実行することで確認できます：

```
specforge validate --antecedent 'label:todo' --consequent 'priority:>50'
```

また、以下のように `specforge.toml` に検証ルールを指定することもできます（[プロジェクト設定](#)を参照）：

```
[[validation_rules]]
antecedent = "label:todo"
consequent = "priority:>50"

[[validation_rules]]
antecedent = "label:blue OR label:green"
consequent = "label:grue"
```

引数なしで `specforge validate` を実行すると、`specforge.toml` で指定されたすべての検証ルールが確認されます。また、VSCode拡張機能は、`specforge.toml` ファイル内の満たされていない検証ルールについて自動的に警告を表示します。

一括ラベル付け

`bulk-label` CLIコマンドを使用すると、クエリに一致するすべての仕様にラベルを追加できます。例えば、以下のコマンドは `priority` が50より大きいすべての仕様に `important` ラベルを追加します：

```
specforge bulk-label --query 'priority:>50' --label 'important'
```

同様に、以下のコマンドは `label:green` または `label:blue` を満たすすべての仕様に `grue` ラベルを追加します：

```
specforge bulk-label --query 'label:green OR label:blue' --label 'grue'
```

コマンドラインインターフェース

CLIには自身のドキュメントが含まれています。 `specforge --help` を実行するとすべての利用可能なコマンドが表示され、 `specforge <command> --help` で各コマンドの詳細な使い方を確認できます。

```
$ specforge --help
Usage: specforge COMMAND [--verbosity VERBOSITY | --verbose]
        [-p|--project PROJDIR]

Available options:
  --verbosity VERBOSITY  Set log verbosity: none, attention, info, trace
                        (default: info)
  --verbose              Alias for --verbosity trace
  -p,--project PROJDIR  Path to the project directory (default: ".")
  --stdio               Compatibility flag for language-client stdio
                        transports (ignored)
  -h,--help             Show this help text
  --version             Show version information

Available commands:
  parse                 Parse a spec file
  check                 Typecheck a spec file
  lint                 Run fast project diagnostics
  analyze               Run solver-backed project analyses
  format                Format a spec file
  monitor               Monitor a spec file
  eval                 Evaluate a spec file
  serve                 Start the server
  lsp-server            Run the LSP server on stdio
  export                Export a spec
  streammon             Monitor a spec in streaming mode from STDIN
  init                  Initialize a new Lilo project
  flatten               Flatten hierarchical components in a system
  schema                Generate a schema or template
  doctor                Troubleshoot common issues
  conditional-inconsistency
                        Analyze conditional inconsistencies in a system
  search                Search for definitions in a project
  validate              Validate that the results of the antecedent query are
                        contained in the results of the consequent query
  bulk-label            Update labels for selected definitions
  doc                   Print SpecForge documentation pages
  specialize            Preview feature: generate one specialized spec per
                        finite enum param combination
  fill-stubs            Fill in definition stubs using the configured LLM
  specialize-system     Preview feature: specialize one component-free system
                        source file
  set-tier              Set the license tier
```

ドキュメント (doc)

SpecForgeのドキュメントはバイナリに埋め込まれており、直接出力できます。

- `specforge doc` は、各ページの1行の概要とともに利用可能なページを一覧表示します。
- `specforge doc <topic>` は、単一のページを出力します (例: `specforge doc lilo-language`)。
- `specforge doc --all` は、すべてのページを連結して出力します。

プロジェクトディレクトリ

CLIコマンドは、プロジェクトディレクトリ (つまり、 `specforge.toml` ファイルを含むディレクトリ) が現在の作業ディレクトリであると想定します。コマンドの末尾に `-p / --project` オプションを指定することで、別のプロジェクトディレクトリを指定できます。

出力言語

一部のCLIコマンドは、人間が読むためのコマンド出力をローカライズします。英語（デフォルト）と日本語がサポートされており、日本語訳はコマンドごとに順次展開されています。

言語は標準的なロケール環境変数から選択されます。たとえば `$LANG` を変更することで言語を切り替えられます:

```
$ LANG=ja_JP.UTF-8 specforge check # 日本語を強制
$ LANG=en_US.UTF-8 specforge check # 英語を強制
```

レコードエンコーディング

いくつかのコマンドは `RECORD_ENCODING` サブコマンド（`flat` または `nested`）を受け付け、レコード型のシグナルとパラメータの表現方法を制御します。`flat` エンコーディングは、フィールドセパレータを設定するための `-s / --separator SEPARATOR` を受け付けます（デフォルト: `"_"`）。

エンコーディングが指定されない場合、デフォルトはファイル形式によって異なります。**CSVの場合は `flat`**、****JSON/JSONLの場合は `nested` ****です。

詳細な説明と例については、データファイルの章の [レコードエンコーディング](#) を参照してください。

トラブルシューティングガイド

Doctorの実行

SMTソルバー（[静的解析](#)で使用）と接続できているかやLLMへのアクセスなどの問題を診断するには、`doctor CLI` コマンドを実行できます。詳細については、`specforge doctor --help` を実行してください。

プロキシのバイパス

Python SDKを使用してSpecForgeクライアントオブジェクトを作成する際（[Python SDKガイド](#)を参照）、サーバーの正しい `base_url` が指定されていることを確認してください。サーバーをローカルで実行している場合は、正しいポートが指定されていることを確認してください。

システムによっては、JupyterノートブックなどのPython環境からSpecForgeサーバーへのリクエストを傍受する [プロキシ](#) が設定されている場合があります。Python SDKからのリクエストがプロキシをバイパスするようにするには、`NO_PROXY` 環境変数にSpecForgeサーバーのアドレス（通常は `localhost`）を追加してみてください。

```
import os

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

MacOS Gatekeeper

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。



この場合、specforge実行ファイルを手動でホワイトリストに追加できます。方法については、[macOSでのセットアップガイド](#)のこのセクションを参照してください。

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 `specforge.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 `specforge.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例探索エンジン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。