SPTG: Symbolic Path-Guided Test Case Generator

Table of content

SPTG overview

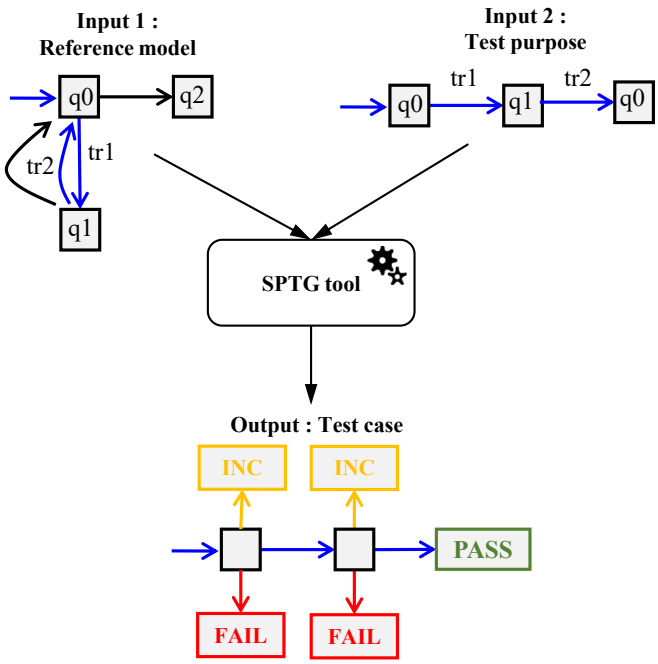


Figure 1: Schematic view of SPTG showing the model automaton with a selected test purpose (blue path) and the generated test case automaton with terminal verdict states.

SPTG is a model-based test generation tool that automatically produces **conformance deterministic test cases** from system models combining both **data** and **timing constraints**. As shown in **Figure 1**, SPTG takes an **automaton model** and a **test purpose**, i.e., a path of the model, and generates the corresponding **test case automaton** with **verdict states** PASS, FAIL, INC (for inconclusive).

It relies on **path-guided symbolic execution**, which explores the input path and builds **symbolic constraints** over inputs and timing. SPTG embeds the **SMT-solver Z3**, which is used to check the **satisfiability of path conditions** along the main test purpose path and its **immediate divergent paths**, as well as to check determinism. Infeasible branches, inconsistent with the test purpose, are pruned early during symbolic exploration, avoiding dead paths that correspond to excluded behaviors.

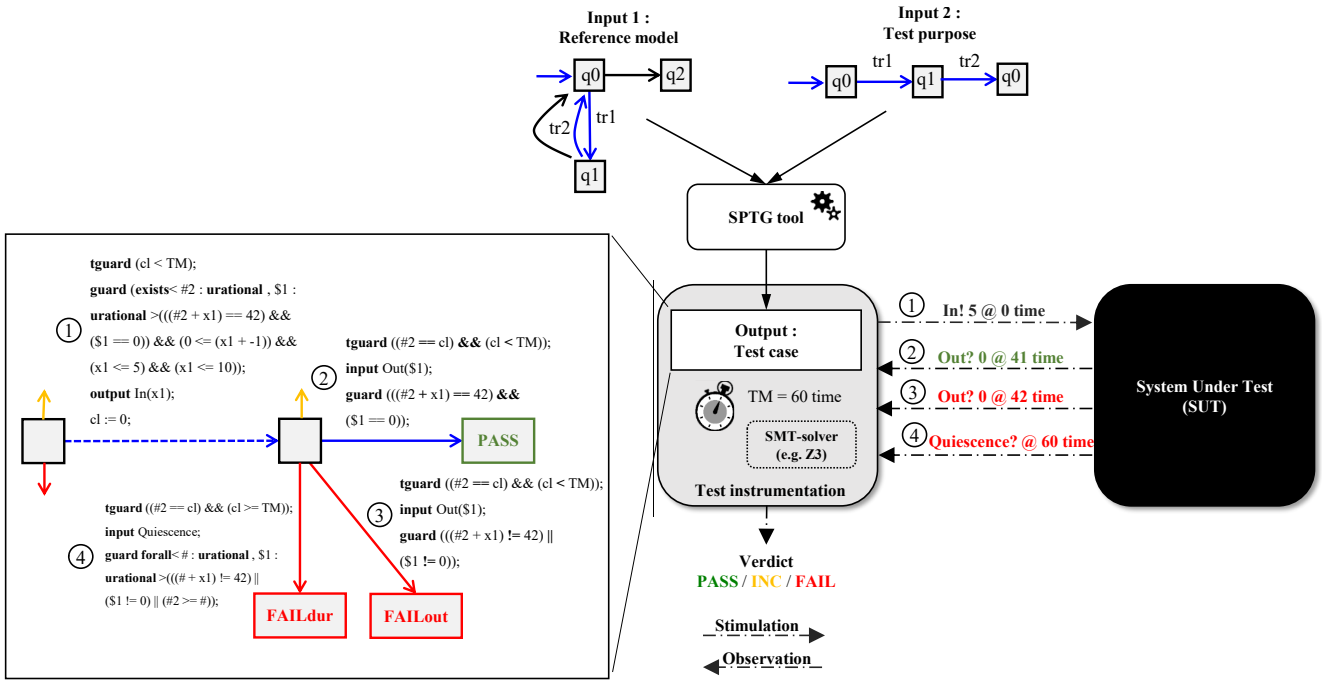


Figure 2: Execution of a generated test case against the System Under Test (SUT) with verdicts determined at runtime.

Figure 2 illustrates the execution phase, where the generated test case interacts with the **System Under Test (SUT)**. During execution, **Z3** is used to solve the **stimulation conditions (guards)**, determining the inputs and timings to apply. Test case transitions are controlled by a clock cl, which satisfies cl < TM, where TM is the maximal waiting time before either applying a stimulation or observing an output. Quiescence, i.e., the obersvation of absence of output, is detected when cl >= TM, indicating that the system remains silent. This timing mechanism, combined with quiescence detection, ensures the test case is implementable in a real-time setting. Additionally, **Z3** is used to check that the **observed outputs** and their **timings** satisfy the corresponding observation conditions, after which verdicts are assigned.

Applications

- **Model-Based Testing (MBT)** of systems exhibiting temporal and data-related behaviors.
- **Offline generation** of efficient and deterministic test cases from formal models.
- **Teaching and demonstration** of symbolic execution and model-based test generation principles.

References

SPTG implements the **symbolic path-guided test generation approach**, developed in: 👉 https://doi.org/10.1016/j.scico.2025.103285 *(Open Access)*.

As an extension of the symbolic execution platform Diversity (https://projects.eclipse.org/proposals/eclipse-formal-modeling-project), which is distributed under the Eclipse Formal Modeling Project, SPTG can leverage its coverage analyses for **test purpose selection**, providing an integrated environment for offline timed symbolic testing.

Quick start with SPTG

SPTG directory Structure:

- `bin/`: This directory contains the SPTG tool binary `sptg.exe`. It also includes the PlantUML JAR which enable visualization and export of generated test cases in SVG format.
- `examples/`: This directory contains all examples. It includes: a subdirectory for each example; and a script `run-all.sh` which executes all preconfigured test case generation tasks across all examples. Each example subdirectory includes:
    - The reference model.
    - A local script (`run-sptg.sh`, `run-sptg-h2.sh`, or `run-sptg-h5.sh`) that calls SPTG for a preconfigured test case generation task.
      The `run-all.sh` script sequentially executes all local scripts.
- `tutorials/`: This directory contains three tutorials and their associated files: a tutorial on model specification, a tutorial on test case generation, and a tutorial on test purpose selection. The test purpose selection feature is inherited from the symbolic execution platform Diversity, which SPTG extends.
- `src/`: Contains the C++ source code of SPTG.
- `third-party/`: Directory for third-party libraries.
- `packages/`: Directory for dependencies.
- `Release/`: Contains release artifacts.
- `LICENSE`: The artifact license (same license as the Diversity symbolic execution platform).
- `examples-outputs.zip`: Compressed folder containing outputs generated by executing all examples from our experiments.
- `README`: This file.

⚠ **Install dependencies (Graphviz, ...)** ⚠

The required dependencies are provided in the `packages` directory.

```
cd /path/to/SPTG/packages
sudo dpkg -i *.deb
```

Start with dummy example

```
cd /path/to/SPTG/examples/example02_dummy/
./run-sptg-h2.sh
```

This script instructs **SPTG** to generate a **test case** with the following specifications:

- **Reference model:**
  `/path/to/SPTG/examples/example02_dummy/example02_dummy.xlia`
- **Test purpose:** Defined as the **sequence of transitions**: `tr1; tr2`
- **Action:** Generate a **test case** corresponding to the given reference model and test purpose.

> **Note:**
> The input reference model automaton is encoded in the **XLIA language** (file `.xlia`), the input language of the symbolic execution platform **Diversity**, which **SPTG** extends. See tutorial on model specification here for more details.

SPTG generates the resulting **test case automaton** in the following formats:

- **Graphical format: PlantUML**
  File `/path/to/SPTG/examples/example02_dummy/output_h2/testcase.puml`
  Comment: This file provides a visual representation of the test case automaton, which can be rendered using PlantUML.
- **JSON format with SMT-LIB guards**
  File `/path/to/SPTG/examples/example02_dummy/output_h2/testcase_smt.json`
  Comment: This JSON file encodes the test case automaton, including guards in SMT-LIB format, suitable for automated execution againt system under test (SUT) using an SMT-solver (e.g. Z3).
- **Specification language: XLIA**
  The same language used to express the reference model.
  File `/path/to/SPTG/examples/example02_dummy/output_h2/testcase.xlia`
  Comment: This file can be explored using the symbolic execution platform Diversity.

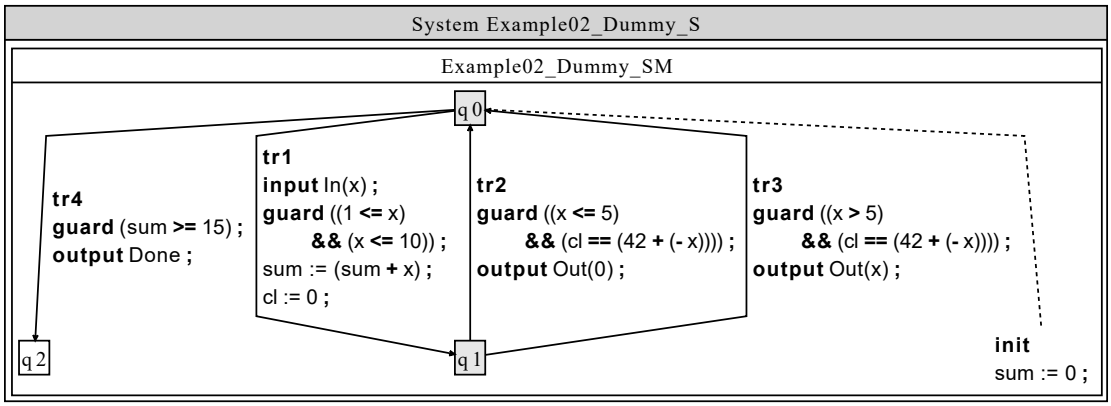
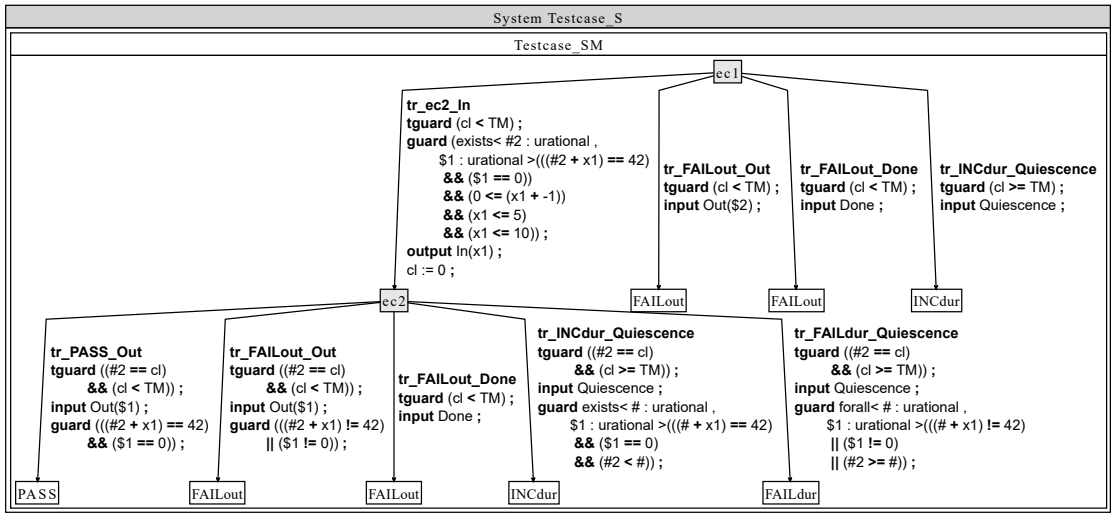
> **Note:** The script also generates the graphical **PlantUML** file for the reference automaton:
> **File** `/path/to/SPTG/examples/example02_dummy/output_h2/example02_dummy.puml`
> *Comment:* This file provides a visual representation of the reference automaton.

> **Note:** You can visualize `.puml` files using PlantUML or the online tool PlantText. You can convert a file `.puml` to a file `.svg` (see the PlantUML Conversion Guide).

> **Note:** If the PlantUML JAR is located in `/path/to/SPTG/bin` and ⚠ Graphviz is installed ⚠, the script automatically produces:
> **File** `/path/to/SPTG/examples/example02_dummy/output_h2/example02_dummy.svg`
> **File** `/path/to/SPTG/examples/example02_dummy/output_h2/testcase.svg`

The table below summarizes the inputs and outputs for generating the **test case** with SPTG. The figures shown are **visual representations** obtained by converting the corresponding **PlantUML** files into **SVG** format.

Description	Content

Description	Content
Input 1: *Reference system model (Timed Symbolic Automaton)*	
Input 2: *Test purpose (Sequence of transitions)*	`tr1; tr2`
Output: *Generated test case (Deterministic Timed Symbolic Automaton)*	

Run all examples

```
cd /path/to/SPTG/examples/
./run-all.sh
```

> **Note:** The results of running all examples are provided in the file `/path/to/SPTG/examples-outputs.zip`.

Compilation instructions

The compilation procedure is detailed below for recent linux.

To compile SPTG, navigate to the `Release` directory:

```
cd Release/
```

Then build the project:

```
make all -j4
```

During compilation, the process automatically overwrites the existing `sptg.exe` in the `bin` directory using:

```
cp -f sptg.exe ../bin/sptg.exe
```

If you wish to preserve the existing executable, rename it before compilation for instance as follows:

```
mv ../bin/sptg.exe ../bin/sptg_old.exe
```

> **Note:** The compilation and testing of SPTG have been performed on the virtual machine published at https://doi.org/10.5281/zenodo.17171929.
> The VM runs on **Ubuntu 25.04**, and the compilation was executed within this environment.
> The VM was executed with **VirtualBox Version 7.1.12 r169651 (Qt 6.5.3)** https://www.virtualbox.org/.

> **Note:** The `-j4` option in the `make` command allows up to 4 compilation jobs to run in parallel, speeding up the build process by using multiple CPU cores.

SPTG tutorials

📓 Tutorial on model specification:

`/path/to/SPTG/tutorials/model_specification.pdf`

📓 Tutorial on test case generation:

`/path/to/SPTG/tutorials/testcase_generation.pdf`

📓 Tutorial on test purpose selection:

`/path/to/SPTG/tutorials/testpurpose_selection.pdf`

PlantUML: PUML to SVG Conversion Guide

A concise reference for converting `.puml` files to `.svg` images via the command line.

Prerequisites

1. **Java Runtime Environment (JRE):** Required to execute PlantUML.
2. **PlantUML JAR File:** The standalone PlantUML application.
3. **Graphviz:** Used internally by PlantUML for layout and rendering.

> **Note:** If ⚠Graphviz is installed ⚠, it will be available in your system path.

Download PlantUML

Get the latest stable release of `plantuml.jar` from:
👉 https://github.com/plantuml/plantuml/releases

Ensure both `java` and `dot`(Graphviz) commands are available:

```
java -version
dot -V
```

Conversion Command

Navigate to the folder containing both `plantuml.jar` and your `.puml` file.

Use the `-tsvg` flag to generate an SVG image:

Command	Action
`java -jar plantuml.jar -tsvg yourfile.puml`	Converts the input file (`.puml`) to an SVG output (`.svg`).

Example

```
# Generates 'MyDiagram.svg'
java -jar plantuml.jar -tsvg MyDiagram.puml
```