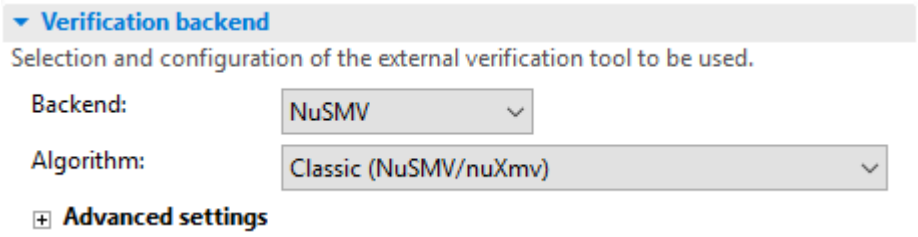
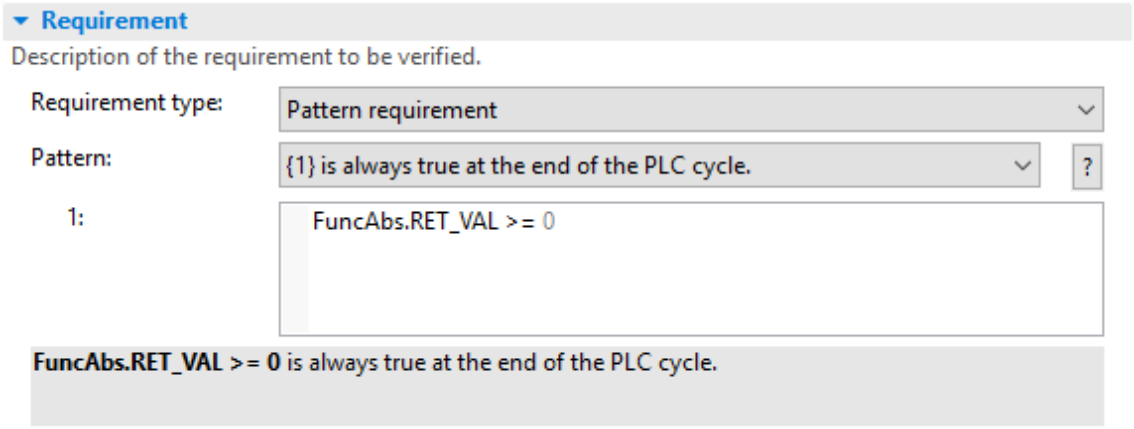
verification.

- The inputs of this block (as well as the input memory area) will be re-initialized using non-deterministic values at the beginning of each PLC cycle. Each other block's input values will be expected to be defined by the callers.
- If the selected block is a function block, a new instance will be created for it.

iii. **Fill the *Verification backend* section.** Here you should select the verification engine (verification backend) to be used to perform this verification case. Depending on the selected verification backend, additional settings may be available.



iv. **Fill the *Requirement* section with a *Pattern-based requirement*.** Here you should provide the requirement that will be checked upon the execution of the verification case. Currently, two types of requirements are supported: assertions and pattern-based requirements.



- For this example, choose the *Pattern requirement* for the *Requirement type* option.
- The *Pattern* combobox shows the supported pattern types.
  - It is possible to define your own requirement patterns and to load them in the *Preferences menu*.
  - You can click on the *?* button to get more information about the available patterns.
  - For this example, choose the "*{1} is always true at the end of the PLC cycle.*" pattern.
- Fill the placeholders of the selected pattern. In this example, there is only one placeholder (`{1}`). Type `FuncAbs.RET_VAL >= 0` to the corresponding textbox.
  - *Ctrl+Space* can be used to open the content assist menu.
  - If the selected entry block was a function block, the automatically created instance would be available under the name `instance` .
  - If you hover the mouse over the `1:` text, you get more information about the expected expression.
- Note that the syntax used to fill the placeholders is defined by PLCverif and is independent from the used language frontend (STEP 7).

  - Type: int (optional)
  - Default value: `5`

- **-job.backend.binary_path**: Full path to the Theta binary (.jar).
  - Type: Path
  - Default value: `.\tools\theta\theta-cfa-cli.jar`
- **-job.backend.lib_path**: Path of the libraries (Z3 DLLs: libz3.dll, libz3java.dll or Z3 SOs: libz3.so, libz3java.so) required by Theta.
  - Type: Path
  - Default value: `.\tools\theta`
- **-job.backend.domain**: Abstract domain. See Theta manual for more details and accepted values.
  - Type: String
  - Default value: `PRED_CART`
- **-job.backend.refinement**: Refinement strategy. See Theta manual for more details and accepted values.
  - Type: String
  - Default value: `BW_BIN_ITP`
- **-job.backend.search**: Search strategy. See Theta manual for more details and accepted values.
  - Type: String
  - Default value: `ERR`
- **-job.backend.precgranularity**: Granularity of the precision of abstraction. See Theta manual for more details and accepted values.
  - Type: String
  - Default value: `LOCAL`
- **-job.backend.predsplit**: Splitting method for new predicates obtained from interpolation. See Theta manual for more details and accepted values.
  - Type: String
  - Default value: `WHOLE`
- **-job.backend.encoding**: Encoding of the CFA. See Theta manual for more details and accepted values.
  - Type: String
  - Default value: `LBE`
- **-job.backend.maxenum**: Maximal successors to enumerate using the SMT solver. See Theta manual for more details and accepted values.
  - Type: String
  - Default value: (empty)
- **-job.backend.initprec**: Initial precision of the abstraction. See Theta manual for more details and accepted values.
  - Type: String
  - Default value: `EMPTY`

Pattern requirement (`-job.req = pattern`)

Represents the requirement based on a selected and filled requirement pattern.

- **-job.req.pattern_file**: Pattern definition XML file to be used. Leave empty to use the built-in patterns.
  - Type: Path (optional)
- **-job.req.pattern_id**: ID of the requirement pattern to be used. (The given permitted values do not apply if not the default patterns are used.)

  - Type: String
  - Permitted values: `pattern-implication`, `pattern-invariant`, `pattern-forbidden`, `pattern-statechange-duringcycle`, `pattern-statechange-betweencycles`, `pattern-reachability`, `pattern-repeatability`, `pattern-leadsto`

- **-job.req.pattern_params**: Parameters to fill the selected pattern. The required number of parameters depends on the selected pattern.
  - Type: String

- **-job.req.inputs**: List of (PLC) variable names to be treated as inputs.
  - Type: List (optional)

- **-job.req.params**: List of (PLC) variable names to be treated as parameters.
  - Type: List (optional)

- **-job.req.bindings**: PLC variable value bindings.
  - Type: RequirementInputBindings (optional)

- **-job.req.bindings.variable**: Variables to be bound.
  - Type: List

- **-job.req.bindings.value**: Values to be used in binding.
  - Type: List

Assertion requirement (`-job.req = assertion`)

Represents the requirement based on the assertions present in the source code.

- **-job.req.inputs**: List of (PLC) variable names to be treated as inputs.
  - Type: List (optional)

- **-job.req.params**: List of (PLC) variable names to be treated as parameters.
  - Type: List (optional)

- **-job.req.bindings**: PLC variable value bindings.
  - Type: RequirementInputBindings (optional)

- **-job.req.bindings.variable**: Variables to be bound.
  - Type: List

- **-job.req.bindings.value**: Values to be used in binding.
  - Type: List

- **-job.req.assertion_to_check**: The tag(s) of the assertion(s) to be checked, or '*' if all assertions shall be checked.
  - Type: List (optional)
  - Default value: `{*}`

- **-job.req.max_expr_size_for_detailed_diagnosis**: Maximum expression size that is considered for detailed diagnosis if the assertion has been violated.
  - Type: int (optional)
  - Default value: `250`

- **-job.req.assert_representation_strategy**: Assertion (error field) representation strategy. If it is 'INT', the violated assertion can be determined, but the state space will be bigger. If it is 'BOOL', the violated assertion cannot be determined (only the fact that an assertion has been violated), but the state space will be smaller.
  - Type: String (optional)
  - Permitted values: `INTEGER`, `BOOLEAN`

- Default value: `INTEGER`

Summary report serializer (`-job.reporters.<ID> = summary`)

Provides the verification summary job. It can collect the results of previous verification jobs and produce a summary report.

HTML reporter (`-job.reporters.<ID> = html`)

Generates a HTML representation of the verification results, including the metadata, configuration and the eventual counterexample. Only applicable if job=verif.

- **-job.reporters.<ID>.min_log_level**: Minimum log message severity to report.
  - Type: String (optional)
  - Permitted values: `Error` , `Warning` , `Info` , `Debug`
  - Default value: `Warning`
- **-job.reporters.<ID>.include_settings**: Include the effective settings in the report.
  - Type: Boolean (optional)
  - Permitted values: `true` , `false`
  - Default value: `true`
- **-job.reporters.<ID>.show_logitem_timestapms**: Show timestamps for the log items.
  - Type: Boolean (optional)
  - Permitted values: `true` , `false`
  - Default value: `false`
- **-job.reporters.<ID>.hide_internal_variables**: Hides the variables generated for verification purposes, which do not have a corresponding variable in the source code.
  - Type: Boolean (optional)
  - Permitted values: `true` , `false`
  - Default value: `true`
- **-job.reporters.<ID>.include_stack_trace**: Includes stack traces in the log when available.
  - Type: Boolean (optional)
  - Permitted values: `true` , `false`
  - Default value: `false`
- **-job.reporters.<ID>.use_lf_value_representation**: Uses hints to provide type-specific value representation in the counterexample.
  - Type: Boolean (optional)
  - Permitted values: `true` , `false`
  - Default value: `false`
- **-job.reporters.<ID>.show_verification_console_output**: Shows the console output of the verification tool.
  - Type: Boolean (optional)
  - Permitted values: `true` , `false`
  - Default value: `true`

Plain text reporter (`-job.reporters.<ID> = plaintext`)

Generates a plain text representation of the verification results, including the metadata, configuration and the eventual counterexample. Only applicable if job=verif.

- **-job.reporters.<ID>.min_log_level**: Minimum log message severity to report.

List of desired features

- **Parsing PLC programs** written in various languages
  - languages:
    - base Siemens (program layout, variable and block declarations, types)
    - Siemens SCL
    - Siemens STL
    - import symbol tables
  - extra features:
    - handling all source files of a project together (cross-file references)
  - features not to be supported:
    - pointers
    - symbol table editing
- **Representing PLC programs** in an intermediate formalism
  - intermediate formalism:
    - implements a CFA in a convenient way
    - allows reductions
    - allows several CFA's interconnection
    - no concurrency, limited modularization (no synchronization)
    - allows (at least partial) reconstruction of the original code structure (annotations about control structures)
- Support for **requirement formalization**
  - various user-friendly representations
    - requirement patterns
    - assertions in the code
    - in the future, tabular requirement description methods may be supported
  - representation in LTL and/or CTL in a *model checker-independent* way
- Support for **external model checkers**
  - support for general-purpose model checkers
    - NuSMV/nuXmv
    - Theta
  - support for software model checkers
    - CBMC
  - full integration, including input generation, execution and output parsing
- **Reporting**
  - human-readable report
  - contains counterexample if available
  - XML report for Jenkins integration

High-level development principles and decisions

- Maintainability
  - The source code of PLCverif shall have high quality.
  - The critical parts of PLCverif shall be covered by adequate tests.
  - The developer documentation of PLCverif shall cover the important high-level principles.
  - The code should be properly documented in code comments.
- Extensibility
  - PLCverif cannot support all possible use cases out of the box, therefore its architecture is designed to be highly extensible.
  - *PLC languages.* First, PLCverif will support Siemens SCL and STL. In future it can be extended to support SFC, Schneider programs, etc.
  - *Model checkers.* First, PLCverif will support NuSMV/nuXmv and Theta. In future it can be extended to support other FSM-based model checkers.
  - *Requirement specification.* First, PLCverif will support requirement patterns and assertions as requirement specification formalism. In future it can be extended to support other textual/tabular/graphical requirement specification methods.

- It will create a temporal requirement representation corresponding to *none of the assertions has been violated* in CTL.

After the execution of the verification backend, the assertion requirement plug-in will try to provide a diagnosis for violated requirements. It will try to determine which assertion was violated and in which cycle. Depending on the chosen assertion representation strategy, this may not be possible.

Settings

The plain text reporter plug-in can be configured using PLCverif settings (typically with the prefix `job.req.`) which will be represented by a `AssertionRequirementSettings` instance. It extends the `AbstractRequirementSettings` class, thus it also contains the settings described above.

Setting name (Type)	Description	Default value
`assertion_to_check` (`String`)	The name of the assertion to be checked, or `*` if all assertions shall be checked.	`*`
`assert_representation_strategy` (`AssertRepresentationStrategy`)	Assertion (error field) representation strategy.	`INTEGER`
`max_expr_size_for_detailed_diagnosis` (`int`)	Maximum expression size that is considered for detailed diagnosis if the assertion has been violated.	`250`

Note:

- Several assertions may have the same name. If multiple assertions have the given name (`assertion_to_check`), all of them will be checked, thus the verification will succeed if *all of them* are satisfied by the program.

Pattern requirement

The verification pattern requirement (`cern.plcverif.library.requirement.pattern`, ID: `pattern`) represents the given pattern-based requirement as temporal logic requirement.

A **requirement pattern** (represented internally by `Pattern`) is a temporal logic expression with pre-defined placeholders. The user can fill these placeholders with Boolean expressions over PLC variables. Each requirement pattern has a human-readable textual representation as well which contain the same placeholders. This way the user can describe a typical temporal logic requirement by filling in the gaps in a textual representation of the requirement pattern,.

Pattern definition

The patterns are described in an XML file, that is by default the `default-patterns.xml` in the `resources/settings` of the plug-in. It is also possible to define an external file containing other patterns. The pattern definition should match the `patterns.xsd` schema definition included in the plug-in.

Patterns included in the plug-in

Currently the following patterns are supported:

Pattern ID	Textual representation	Formal representation
pattern-implication	If *{1}* is true at the end of the PLC cycle, then *{2}* should always be true at the end of the same cycle.	AG((*{PLC_END}* AND (*{1}*)) --> (*{2}*))
pattern-invariant	*{1}* is always true at the end of the PLC cycle.	AG(*{PLC_END}* --> (*{1}*))
pattern-forbidden	*{1}* is impossible at the end of the PLC cycle.	AG(*{PLC_END}* --> NOT (*{1}*))
pattern-statechange-duringcycle	If *{1}* is true at the beginning of the PLC cycle, then *{2}* is always true at the end of the same cycle.	AG((*{PLC_START}* AND (*{1}*)) --> A[NOT *{PLC_END}* U *{PLC_END}* AND (*{2}*)])
pattern-statechange-betweencycles	If *{1}* is true at the end of cycle N and *{2}* is true at the end of cycle N+1, then *{3}* is always true at the end of cycle N+1.	G((*{PLC_END}* AND (*{1}*) AND X([NOT *{PLC_END}* U (*{PLC_END}* AND (*{2}*))])) --> X([NOT *{PLC_END}* U (*{PLC_END}* AND (*{3}*))]))
pattern-reachability	It is possible to have *{1}* at the end of a cycle.	EF(*{PLC_END}* AND (*{1}*))
pattern-repeatability	Any time it is possible to have eventually *{1}* at the end of a cycle.	AG(EF(*{PLC_END}* AND (*{1}*)))
pattern-leadsto	If *{1}* is true at the end of a cycle, *{2}* was true at the end of an earlier cycle.	NOT (E[(*{PLC_END}* --> NOT (*{2}*)) U (*{PLC_END}* AND *{1}*)])

Settings

The plain text reporter plug-in can be configured using PLCverif settings (typically with the prefix `job.req.`) which will be represented by a `AssertionRequirementSettings` instance. It extends the `AbstractRequirementSettings` class, thus it also contains the settings described above.

Setting name (Type)	Description	Default value
`pattern_id` (`String`)	ID of the pattern to be checked.	--- (mandatory)
`pattern_params` (`List<String>`)	Parameters (expressions) to be substituted in the placeholders of the pattern. The number of given parameters shall match the number of placeholders in the selected pattern.	*empty list*
`pattern_file` (`Path`)	Pattern definition XML file to be used. Leave empty to use the built-in patterns.	`null`

Note:

- The `pattern_id` shall match one of the pattern IDs above if no `pattern_file` is given, or one of the patterns in the provided `pattern_file`.

Example pattern definition file

```
<?xml version="1.0" encoding="UTF-8"?>
<patterns>
  <pattern id="pattern-implication">
    <description>Implication</description>
    <tlexpr type="ctl">AG(({PLC_END} AND ({1})) --> ({2}))</tlexpr>
    <humanreadable>If {1} is true at the end of the PLC cycle, then {2} should always be true at the end of the
    <parameter key="1">
        <description>Condition</description>
    </parameter>
    <parameter key="2">
        <description>Implication</description>
    </parameter>
  </pattern>

  <pattern id="pattern-invariant">
    <description>Invariant</description>
    <tlexpr type="ctl">AG({PLC_END} --> ({1}))</tlexpr>
    <humanreadable>{1} is always true at the end of the PLC cycle.</humanreadable>
    <parameter key="1">
        <description>Invariant to be respected at the end of each cycle.</description>
    </parameter>
  </pattern>
</patterns>
```

Notes:

- The pattern parameters can be defined using the `parameter` tag. The key should be an integer.
- The temporal logic expression (`tlexpr`) should follow the textual syntax of the expression parser (see its grammar).
  - It has to refer to the parameters using `{key}` .
- The `PatternUtils.validate()` method provides validation for a parsed pattern. See its API documentation for the checked features.

- OR (`OR` , `∨`)

```
enum OrBinaryLogicOperator returns expr::BinaryLogicOperator:
    OR='OR' | OR='∨';
```

XorBinaryLogicOperator (enum)

XOR logic operator.

Literals:

```
enum XorBinaryLogicOperator returns expr::BinaryLogicOperator:
    XOR='XOR';
```

ImpliesBinaryLogicOperator (enum)

Implication logic operator.

Literals:

- IMPLIES (`-->` , `→` , `⇒`)

```
enum ImpliesBinaryLogicOperator returns expr::BinaryLogicOperator:
    IMPLIES='-->' | IMPLIES='→' | IMPLIES='⇒';
```

EqualityOperator (enum)

Equality or non-equality comparison operator.

Literals:

- EQUALS (`=`)
- NOT_EQUALS (`<>`)

```
enum EqualityOperator returns expr::ComparisonOperator:
    EQUALS='=' | NOT_EQUALS='<>';
```

ComparisonOperator (enum)

Comparison operator. This does not contain the "equals" and "not equals" operators.

Literals:

- GREATER_EQ (`>=` , `≥`)
- GREATER_THAN (`>`)
- LESS_EQ (`<=` , `≤`)
- LESS_THAN (`<`)

```
enum ComparisonOperator returns expr::ComparisonOperator:
    LESS_THAN='<' | GREATER_THAN='>' | LESS_EQ='<=' | LESS_EQ='≤' | GREATER_EQ='>=' | GREATER_EQ='≥';
```

AdditionOperator (enum)

Addition-type arithmetic operator (addition or subtraction).

Literals:

- MINUS (`-`)
- PLUS (`+`)