

awl2scl — AWL/STL → SCL Konvertierung

Werkzeuge zur (halb-)automatischen Umwandlung der Siemens-AWL/STL-Bausteine dieses Projekts nach SCL für den Umstieg auf TIA Portal V20.

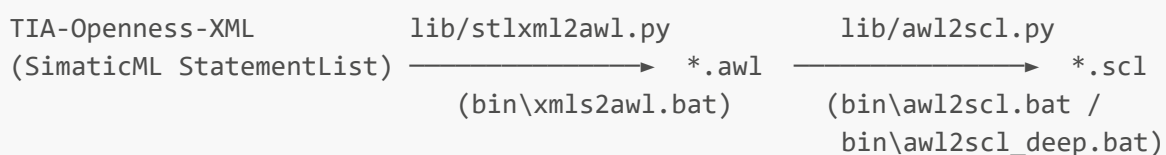
- **lib/stlxml2awl.py** — wandelt TIA-Openness-XML (SimaticML `StatementList`) in lesbaren AWL-Text (`.awl`) um und schätzt je Baustein eine Aufwandskategorie A/B/C. Aufruf über `bin\xmls2awl.bat`.
- **lib/awl2scl.py** — wandelt `.awl`-Dateien mechanisch nach `.scl` um, optional mit Lesbarkeits- und Parameter-Optimierung. Aufruf über `bin\awl2scl.bat` bzw. `bin\awl2scl_deep.bat`.
- **lib/sclopt.py** — SCL-Zwischenstruktur (IR) und die `--improve-gotos`-Durchläufe.
- **lib/paramize.py** — die `--globals2params`-Umwandlung samt persistenter toChange-Liste für die Aufrufer.

Wichtig: Das Werkzeug erzeugt syntaktisch zielgerichteten, mechanisch verhaltensgleichen SCL-Text — **keine** in TIA verifizierte Korrektheit. Die Freigabe (Kompilat in TIA Portal V20, PLCSIM-Vergleich AWL↔SCL) bleibt ein manueller Schritt pro Baustein (siehe [SCL-Export/README.md](#)).

Inhaltsverzeichnis

- [Datenfluss](#)
- [Kategorien A/B/C](#)
- [bin-Skripte](#)
- [CLI-Optionen](#)
- [--improve-gotos](#)
- [--globals2params](#)
- [Schutz handgeschriebener Dateien](#)
- [Log & Ausgaben](#)

Datenfluss



`awl2scl.py` liest **zwei** `.awl`-Formate transparent ein:

1. **gerendert** — von `stlxml2awl.py` erzeugt, Kopf `=== FC 181: Name [STL] ===`.
2. **nativ** — klassische STEP7/TIA-AWL-Quelle, Kopf `FUNCTION_BLOCK "Name", BEGIN/NETWORK`, deutsche Mnemonik (`U/UN/O/SPA/SPB/AUF ...`).

Kategorien A/B/C

Die Kategorie wird je Datei **selbst** bestimmt (nicht aus einem Log übernommen), mit denselben Kriterien wie in `stlxml2awl.py` (Namensmuster aus [SCL-Export/README.md](#) + Sprung-/Pointer-Analyse):

Kat.	Bedeutung	Behandlung
A	einfache Verknüpfungslogik, mechanisch übersetzbar	wird übersetzt
B	Ablaufsteuerung / viele Sprungmarken	nur mit <code>--category B</code>
C	Pointer-/Register-Indirektzugriffe	nur mit <code>--category C</code>

Bausteine, die trotz passender Kategorie ein technisch nicht sicher übersetzbares Konstrukt enthalten (Timer/Zähler-Mnemonic, `OPN DB[var]`, registerindirekte Operanden, unbekannte Sprünge), werden **nicht** übersetzt, sondern im Log unter „**Manuelle Prüfung nötig**“ mit Grund gelistet.

bin-Skripte

Skript	Zweck
<code>bin\xmls2awl.bat</code>	XML → <code>.awl</code> (+ Kategorie-Log)
<code>bin\awl2scl.bat</code>	<code>.awl</code> → <code>.scl</code> , mechanisch (Argumente durchgereicht)
<code>bin\awl2scl_deep.bat</code>	<code>.awl</code> → <code>.scl</code> mit <code>--improve-gotos</code> <code>--globals2params</code>

Beispiele:

```
REM Nur Kategorie A, mechanisch nach SCL-Export\<Station>\
bin\awl2scl.bat --dir . --category A

REM Tiefenlauf: Kontrollfluss aufräumen + Globale zu Parametern
bin\awl2scl_deep.bat --dir . --category A
```

CLI-Optionen (`lib/awl2scl.py`)

Option	Wirkung
<code><datei.awl> ...</code>	einzelne Dateien
<code>--dir <verz></code>	rekursiv alle <code>.awl</code>
<code>--category A[,B,C]</code>	zu übersetzende Kategorien (Default A)
<code>--target-dir <pfad></code>	Zielordner (Default <code>\$PV_SCL_EXPORT</code> bzw. <code><Projekt>\SCL-Export</code>)
<code>--improve-gotos</code>	Kontrollfluss aufräumen (verlustfrei)
<code>--globals2params</code>	globale Merker → Parameter + toChange für Aufrufer
<code>--force</code>	vorhandene generierte <code>.scl</code> überschreiben
<code>--tochange <pfad></code>	toChange-Liste (Default <code><Projekt>\data\globals2params_tochange.json</code>)

Option	Wirkung
<code>--log <pfad></code>	Log-Datei
<code>--show-missing</code>	am Ende alle nicht übersetzten Bausteine (manuelle Prüfung + Fehler) mit Grund auf der Konsole auflisten
<code>--improve-gotos</code> und <code>--globals2params</code> implizieren <code>--force</code> (Regenerierung). Der Zielpfad je Baustein ist <code><target-dir>\<Station>\<Name>.scl</code> .	

--improve-gotos

Räumt den mechanisch erzeugten Kontrollfluss **verlustfrei** auf. Die Übersetzung läuft dazu über eine typisierte Zwischenstruktur (IR, `lib/sclopt.py`); ohne den Schalter ist die Ausgabe byte-identisch zur rein mechanischen Variante. Durchläufe:

- `merge_adjacent_ifdo` — aufeinanderfolgende **IF**-Blöcke mit identischer Bedingung zusammenfassen (aus dem AWL-Idiom `S x; S y; JC L` entsteht sonst dieselbe Bedingung mehrfach).
- `guard_skip_to_if` — `IF c THEN GOTO L; ... ; L: → IF NOT c THEN ... END_IF.`
- `synth_case` / `synth_case_across_networks` — Verteilung auf **eine** Variable gegen Konstanten → **CASE** (auch über REGION-Grenzen hinweg; die REGION-Titel werden zu Branch-Kommentaren).
- `drop_unused_labels` — nicht mehr angesprungene Marken entfernen.
- `normalize_bool` — überflüssige Klammern und **NOT**-Verschachtelungen vereinfachen (`NOT (a <> b) → a = b`).

Nicht sauber auflösbare Sprünge (Rücksprünge, gemeinsame Sammel-Austrittsmarken) bleiben als strukturierter **GOTO** erhalten — die Passes greifen nur beim sicher erkannten Muster.

Beispiel 1 — **HWtoAMRZielFC** (FC 181): Guard-Sprünge → **CASE**

Zwei Netzwerke prüfen dieselbe Variable **"MW190"** gegen Konstanten. Mechanisch:

```

REGION Schleuse 3004: (BST 3)
  IF ("MW190" <> 3004) THEN GOTO X3004E; END_IF;
  "M180.6" := TRUE;
  RETURN;
  X3004E: ;
END_REGION

REGION Schleuse 3565: (BST 4)
  IF ("MW190" <> 3565) THEN GOTO X3565E; END_IF;
  "M180.6" := TRUE;
  RETURN;
  X3565E: ;
END_REGION

```

Mit `--improve-gotos` (Guard→IF + netzwerkübergreifende **CASE**-Synthese):

```

REGION CASE "MW190"
CASE "MW190" OF
  3004:    // Schleuse 3004: (BST 3)
    "M180.6" := TRUE;
  3565:    // Schleuse 3565: (BST 4)
    "M180.6" := TRUE;
END_CASE;
END_REGION

```

Beispiel 2 — **MBlinken** (FB): mehrfache Bedingung zusammenfassen

Das AWL setzt mit *einer* Verknüpfung drei Merker und springt (R x; R y; R z; SPB L). Mechanisch entsteht viermal dieselbe Bedingung:

```

REGION Anzeige bei Error 1-10 1-10 mal blinken lassen
  IF (((((((NOT (#Error_1) AND NOT (#Error_2)) AND ...) AND NOT (#Error_10))))
THEN #hAnzeige := FALSE; END_IF;
  IF (((((((NOT (#Error_1) AND NOT (#Error_2)) AND ...) AND NOT (#Error_10))))
THEN #Flanke_Ausschalttakt := FALSE; END_IF;
  IF (((((((NOT (#Error_1) AND NOT (#Error_2)) AND ...) AND NOT (#Error_10))))
THEN #Kein_Blinken := FALSE; END_IF;
  IF (((((((NOT (#Error_1) AND NOT (#Error_2)) AND ...) AND NOT (#Error_10))))
THEN GOTO NoAn; END_IF;

```

Mit **--improve-gotos** (**merge_adjacent_ifdo** + Klammer-Normalisierung):

```

REGION Anzeige bei Error 1-10 1-10 mal blinken lassen
  IF (((((((NOT (#Error_1) AND NOT (#Error_2)) AND ...) AND NOT (#Error_9)) AND
NOT (#Error_10) THEN
    #hAnzeige := FALSE;
    #Flanke_Ausschalttakt := FALSE;
    #Kein_Blinken := FALSE;
    GOTO NoAn;
  END_IF;

```

(Das verbleibende **GOTO NoAn** zeigt auf eine gemeinsame Austrittsmarke und bleibt bewusst als strukturierter Sprung erhalten.)

--globals2params

Baut **globale Merker-Zugriffe** eines Bausteins in echte Parameter um und verwaltet eine persistente **toChange-Liste** (`data\globals2params_tochange.json`) für die Anpassung der Aufrufer — auch solcher, die noch als `.awl` (nicht konvertiert) vorliegen.

Ablauf je Baustein (`lib/paramize.py`):

1. **Richtungsanalyse:** jedes Global ("MW190", "M180.6", Mx.y, Ex, Ax ...) wird als **Input** (nur gelesen), **Output** (nur geschrieben) oder **InOut** (beides, bzw. uneindeutig — konservativ) eingestuft.
2. **Parametername** deterministisch aus dem Global-Token ("MW190" → MW190, "M180.6" → M180_6), Kommentar // war "...". Semantische Umbenennung bleibt manuell.
3. **Rumpf umschreiben:** jeder Global-Zugriff wird #Param.
4. **Aufrufer finden** (direkte CALL "B" und indirekte UC/CC FC[...]) über alle .awl-Quellen und je Aufrufer den parameterbasierten Aufruf vormerken.

Verhaltenserhaltend by construction: Jeder Aufrufer forwardet exakt dasselbe Global an den Parameter (**Input** mit :=, **Output/InOut** mit =>), solange er das Global vor/nach dem Aufruf wie bisher nutzt.

Grenze: indirekte Aufrufe (UC/CC FC[var], z.B. die CASE-Verteiler in MHMOmniSchleuse) können nicht automatisch umgeschrieben werden → in toChange als **manual** markiert.

Beispiel 1 — **HWtoAMRZielFC**: globale Merker werden Parameter

--globals2params (hier mit --improve-gotos kombiniert) erzeugt aus dem Baustein, der "MW190" liest und "M180.6" schreibt:

```
FUNCTION "HWtoAMRZielFC" : Void
  VAR_INPUT
    MW190 : Word;    // war "MW190"
  END_VAR
  VAR_OUTPUT
    M180_6 : Bool;   // war "M180.6"
  END_VAR
BEGIN
  REGION CASE #MW190
    CASE #MW190 OF
      3004:    // Schleuse 3004: (BST 3)
        #M180_6 := TRUE;
      3565:    // Schleuse 3565: (BST 4)
        #M180_6 := TRUE;
    END_CASE;
  END_REGION
  ...

```

HWtoAMRZielFC wird nur **indirekt** (über UC FC[...]-Verteiler) aufgerufen — daher ein **manual**-Eintrag in der toChange-Liste:

```
"=A01+UH07-KF00|DatenIn": {
  "path": "...\\DatenIn.awl",
  "changes": [
    { "func": "HWtoAMRZielFC",
      "call": "// MANUELL: indirekter Aufruf von HWtoAMRZielFC auf Parameter
umstellen",
      "status": "manual" }
  ]
}
```

```
]
}
```

Beispiel 2 — direkter Aufrufer wird automatisch umgeschrieben (2-Phasen)

Ein Baustein **Sub_Callee**, der "MW50" liest und "M60.0" schreibt, und ein direkter Aufrufer **Top_Caller** (CALL FC "Sub_Callee").

Lauf 1 — **Sub_Callee** wird parametrisiert; der Aufruf-Rewrite wird als **pending** in `data\globals2params_tochange.json` vermerkt:

```
FUNCTION "Sub_Callee" : Void
  VAR_INPUT
    MW50 : Word;    // war "MW50"
  END_VAR
  VAR_OUTPUT
    M60_0 : Bool;   // war "M60.0"
  END_VAR
BEGIN
  REGION test
    #M60_0 := (#MW50 >= 100);
  END_REGION
```

```
"_|Top_Caller": {
  "changes": [
    { "func": "Sub_Callee",
      "call": "\"Sub_Callee\"(M60_0 => \"M60.0\", MW50 := \"MW50\");",
      "status": "pending" }
  ]
}
```

Lauf 2 — beim Konvertieren von **Top_Caller** wird der vorgemerkte Rewrite angewandt (Status **pending** → **applied**); aus dem mechanischen `"Sub_Callee"()`; wird der parameterbasierte Aufruf:

```
REGION ruft callee
  "Sub_Callee"(M60_0 => "M60.0", MW50 := "MW50");
END_REGION
```

So bleibt die toChange-Liste über mehrere Konvertierungsläufe hinweg gültig, bis jeder Aufrufer konvertiert und sein Aufruf angepasst wurde.

Schutz handgeschriebener Dateien

Generierte `.scl` tragen die Kopfzeile `// Mechanische AWL->SCL-Konvertierung (awl2scl.py)`. Mit `--force` werden **nur** Dateien mit diesem Marker überschrieben; handgeschriebene Piloten (z.B. `SCL-Export\=A01+UH06-KF00\MZylinder.scl`, `MHMOmniSchleuse.scl`, `FifoIn.scl`, `DatenOut.scl`) bleiben unangetastet und erscheinen im Log unter „**Geschützt, handgeschriebene .scl nicht überschrieben**“.

Log & Ausgaben

Jeder Lauf schreibt `log\awl2scl_<Zeitstempel>.log` mit:

- Kopfstatistik (angeforderte Kategorien, aktive Schalter, Zähler),
- **Manuelle Prüfung nötig** (Baustein + Grund),
- **Parametrisiert** (neue Parameter je Baustein + Anzahl Aufrufer/davon manuell),
- **GOTO-lastig** (viele Sprünge, nur ohne `--improve-gotos`),
- **Geschützt / Übersprungen** (Kategorie bzw. Ziel existiert),
- **Fehler**.

Bei `--globals2params` zusätzlich der Pfad der toChange-Liste.

Mit `--show-missing` wird am Ende zusätzlich auf der **Konsole** (nicht nur im Log) eine kompakte Liste aller nicht übersetzten Bausteine ausgegeben — sowohl **manuelle Prüfung nötig** als auch echte **Fehler**, je mit Grund:

```
=====
Nicht uebersetzt (2)
=====
MANUELL [B] =A01+UH06-KF00\Programmbausteine\Anlage\AMR_Bereich.awl
      Grund: Timer-Mnemonic SE nicht unterstuetzt
MANUELL [C] =A01+UH06-KF00\Programmbausteine\Module\Set_Input.awl
      Grund: registerindirekter/zeigerbasierter Operand: = E[ #AR1Zeiger]
```

Bausteine, die nur wegen falscher Kategorie oder als reiner Datenbaustein übersprungen wurden, zählen nicht als „nicht übersetzt“ (die stehen bereits in den eigenen Log-Abschnitten).