Karlsruhe Institute of Technology

Regression Verification for Programmable Logic Controller Software

Master Thesis of

Alexander Sebastian Weigl

At the Department of Informatics
Institute for Theoretical Informatics

Reviewer:	Prof. Dr. rer. nat. Bernhard Beckert
Advisor:	Dr. rer. nat. Mattias Ulbrich

Duration:: 16. July 2014 – 15. January 2015

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 15. 01. 2015

.....

(**Alexander Sebastian Weigl**)

Abstract

Plants are a long-term and expensive investment, additionally, they need to be adapted to the market and technology process. The plant evolves by changing of hardware and software components. A evolution step can introduce defects in the plant workflow. How can we find such introduction of defects in view of the high safety claim of plants? In this master thesis we apply regression verification on two version of automation software, to prove the equivalence or to find divergence in the software behaviour. We investigate the regression verification on automation software for Programmable Logic Controller (PLC) according to the norm IEC 61131-3, especially by using the software from *Pick-and-Place-Unit* (PPU) case study [VH+14]. We define the equivalence of PLC software and make observation for the soundness of equivalence proofs. Notably, we introduce the *conditional equivalence*. Besides theoretical aspects we provide tool chain for the regression verification of Structured Text (ST) and Sequential Function Chart (SFC). We formalize syntax and semantic for ST and SFC, and define an intermediate language ST_0, as a subset of ST. The transformation covers only unwindable loops and unfoldable data structures. The equivalent ST_0 consists only one program, only with assignments and if statements. We translate ST_0 into the Symbolic Model Verifier (SMV) format with symbolic execution via Single Static Assignment (SSA). From the experience of finding the equivalence condition in the case study we develop an approach for deriving conditions from the bisimulation of SFCs. Moreover, we consider high level user-friendly operators for writing these conditions. We proof equivalence between the case study scenarios, by restricting the program state and input space. We use the nuXmv model checker and IC3 to validate the invariants claiming the equivalence. The runtime span from seconds in easy cases, till couple of hours or even days, but we show that regression verification is feasible upon automation software. This is the first published work on regression verification and equivalence check of automation software.

Zusammenfassung

Einleitung

Fabrikanlagen sind eine Investition über lange Zeiträume. Durch den Preisdruck des Marktes und technische Erneuerungen müssen Änderungen an den Bauteilen und der Steuerungssoftware vorgenommen werden. Die Anlage erreicht neue *Evolutionsstufen* um sich an die neuen Situationen anzupassen. Die Evolutionsstufe stellt eine neue Kombination aus Bauteilen und Software dar. Wie können wir sicher stellen, dass der Evolutionschritt kein unerwünschtes Verhalten im Betrieb der Anlage einführt? Unerwünschte Verhalten entstehen durch eingeführte Fehler in den Bauteilen oder Software oder durch falsche Abstimmung zwischen der Hard- und Software. Unter Berücksichtigung, dass von Industrieanlagen Gefahr von Leib und Leben ausgehen können, ist eine zuverlässige Erkennung unerlässlich.

Regression Testing ist ein Verfahren aus dem Softwareengineering, dass für die Erkennung des Auftauchens von Fehlern eingesetzt wird. Hierzu wird wie bei Softwaretests, die Ausgabe der neuen Softwarerevision gegen eine Referenzausgabe abgeglichen. Hat sich die Ausgabe zwischen zwei Softwarerevisionen geändert, haben wir ein anderes Verhalten in der Anlage entdeckt. Zum Testen halten wir eine definierte Menge von Eingabeninstanzen vor und nur für diese Eingabeinstanzen gilt die Aussage, dass die Anlage nach dem Softwareupdate das äquivalente Verhalten zeigt. Mit Maßnahmen aus der Verifikation möchten wir zeigen, dass das Verhalten für alle Eingabeinstanzen äquivalent ist.

Die Äquivalenz zwischen Fabrikanlagen ist eine vielschichtige Aussage. Wir müssen den Begriff schärfen. Zum Einen besteht die Anlage aus verschiedenen Schichten wie Software, Steuerungshardware, Stellglieder und Sensoren und einem Bussystem zur Kommunikation. Zum Anderen passiert eine Evolution mit Absicht um Änderungen an der Software und Hardware, zum Beispiel das Beheben von Fehlern oder die Unterstützung von neuen Sensoren. Wenn wir die Äquivalenz zwischen zwei Revisionen zeigen möchten, müssen wir das geänderte Verhalten außer Acht lassen. Als Letztes ist auch eine Anpassung der eventuell geänderten Eingabe- und Ausgabevariablen zwischen beiden Softwarerevisionen nötig. Die Äquivalenz unterliegt verschiedenen Abstraktionsebenen, die eine verschieden starke Aussage über die Äquivalenz zulassen.

Aufgabenstellung

In dieser Masterarbeit behandeln wir die Regression Verification zwischen zwei Softwareversionen für die Steuerung von Fabrikanlagen. Die Steuersignale versendet der *Programmable Logic Controller* nach der Ausführung der Steuerungssoftware mit den aktuellen Sensordaten. Wir fokussieren uns dabei auf die zyklischen Arbeitsmodi der PLC. Als Evaluationsgrundlage bietet *Pick-and-Place-Unit* Fallstudie (PPU) von der Technischen Universität München [VH+14], verschieden-artige Softwareevolution für die PPU-Anlage. Ziel der Arbeit ist die Erarbeitung der theoretischen Grundlagen für die Regression Verfication zwischen Automatisierungsoftware, sowie Führung der Äquivalenzbeweise zwischen den verschiedenen Szenarien der PPU. Wir bauen dafür eine Kette von Operatoren, um die zwei Sprachen *Structured Text* (ST) und *Sequential Function Chart* (SFC) in Modell für den Modellchecker zu übersetzen. Insgesamt definiert IEC 61131-3 fünf möglichen interoperatiblen Programmiersprachen für die Programmierung von PLCs.

Theorie

Fabrikanlagen stellen ein cyber-physisches System dar. Die PLC-Software steuert über

die Stellglieder die physischen Größen der Anlage, wie Geschwindigkeit von Werkstücken, pH-Werte über Mischung von Flüssigkeiten oder Temperaturen. Dafür sendet die PLC Kommandos über ein Bussystem an die Stellglieder, die dann wiederum intern eine Logik besitzen, um die Spannungen und Ströme der Leistungselektronik zu regeln. Messinstrumente nehmen die physischen Größen auf und verarbeiten diese für Eingabe in die PLC vor. Wir haben damit eine Feedbackschleife, die PLC erhält die Messwerte von physischen Größe, trifft Entscheidungen und lässt diese Größe über die Stellglieder ändern.

Für unsere Beobachtungen für die Äquivalenz von PLC-Software spielt die Umgebung der PLC zwei wichtige Rollen. Zum Einen müssen wir die Umgebung für die *Soundness*-Betrachtungen unseres Äquivalenzbeweise berücksichtigen. Zum Anderen helfen berechenbare Modelle der Anlage den möglichen Raum von Eingabegröße einzuschränken.

Im Fokus dieser Arbeit stellen wir die bedingte Äquivalenz (*conditional equivalence*) vor. Die bedingte Äquivalenz schließt bestimmte Ereignisse in der Gleichheit aus und erlaubt auch eine partielle Gleichheit der Ausgabewerte. Zum Beispiel sind beide Softwareversionen im Ausgabeverhalten äquivalent, wenn keine metallischen Werkstücke vorkommen. Eine solche Bedingung bezieht sich nachher auf die konkreten Messwerte von Sensoren und Variablenbelegung im Speicher der PLC. Häufig lässt sich die Bedingung stärker formulieren, zum Beispiel haben die beiden Softwareversionen ein äquivalentes Verhalten bis zum Auftreten von metallischen Werkstücken. Die partielle Gleichheit der Ausgabewerte erzwingt eine Betrachtung über Soundness. Wir vergleichen beide Softwareversionen in einer Bisimulation bei gleichen Eingaben in jedem Zyklus der PLC. Ist die Ausgabe der Softwareversionen unterschiedlich, muss eine Rechtfertigung für Annahme der identischen Eingaben erfolgen. Eine solche Rechtfertigung ist zum Beispiel die fehlende Rückkopplung der von der Gleichheit ausgelassenen Stellgliedvariablen.

Methodik Wir wandeln die PLC-Software in SMV Modelle um. Symbolic Model Verifier (SMV) ist ein expliziter Modellchecker. Wir verwenden einen der Nachfolger *nuXmv*, um mit IC3 (*Incremental Construction of Inductive Clauses for Indubitable Correctness*) [McM03] Invarianten zu überprüfen, die die Gleichheit beider Softwareversionen sicherstellen. IC3 versucht eine Menge von Klauseln zu finden, die induktiv zum Transitionssystem und aus denen die Safety Property folgt.

Die Umwandlung von PLC-Software geschieht mehrstufig. Wir normalisieren SFC, durch Auslagerung von simultanen Abschnitte in SFCs. Die normalisierten SFC_0 werden in ST codiert. ST_0 ist eine reduzierte Variante von ST. Wir erhalten ST_0 durch Code-Transformationen aus ST

- Ausrollen von FOR-Schleifen,
- Auspacken von Arrays und Strukturen,
- Timer-Abstraktion,
- Einbettung von Unterprozeduren (*function blocks*).

ST_0 ist frei von Schleifen oder komplexeren Datentypen und enthält nur eine Prozedurenrumpf.

Für die Formulierung der Bedingungen für die Äquivalenz definieren wir vier Operatoren UNTIL_ϕ, AFTER_α, $\mathrm{WITHIN}^\omega_\alpha$ und $\mathrm{EXCEPT}^\omega_\alpha$. Mit UNTIL_ϕ fordern wir eine Gleichheit zwischen beiden Softwareversionen bis ϕ eintrifft. Umgekehrt wird bei AFTER_α die Gleichheit erst geprüft nachdem α wahr war. $\mathrm{WITHIN}^\omega_\alpha$ fordert Gleichheit innerhalb der Grenzen, die mit α und ω definiert werden, sowie $\mathrm{EXCEPT}^\omega_\alpha$ lässt die Forderung nach der Gleichheit in den Grenzen fallen. Die Operatoren dienen zur lesbarer Notation für unsere Bedingungen in der PPU Fallstudie und sollen die Definition von Bedingungen für Ingenieure vereinfachen.

Aus unseren Erfahrungen über die Herleitung der Bedingungen für der Fallstudie, leiten wir eine Heuristik zur Ermittlung von Bedingungen für die Äquivalenz ab. Eine Bisimulation von zwei SFC liefert die Stellen, die zu einer Abweichung in den Stellgliedern führt. Die abgeleitete Bedingung schließt diese Abweichungen aus der Äquivalenz aus.

Ergebnisse

Die PPU Fallstudie enthält 14 verschiedene Softwareversionen. Wir haben die Äquivalenz zwischen den einzelnen Version bewiesen. Dazu benötigt jeder Beweise, eine Bedingung für die Äquivalenz die von der Änderung im Evolutionschritt abhängig ist. Die Bedingungen haben wir manuell durch Vergleichen der Software hergeleitet und mit den Änderungen an der Anforderung aus [VH+14] abgeglichen. Laufzeiten der Beweise reichen von wenigen Sekunden bis Stunden, je nach Komplexität der Unterschiede und der gestellten Bedingung.

Related Work

Diese Arbeit verbindet die Regression Verification und die Verifikation in der Automatisierungsoftware. Regression Verification ist Bestandteil von [GS09; GS13; Str09]. In der Kategorisierung von [GS08] ist unsere bedingte Äquivalenz als *Reactive equivalence* einzuordnen. Die Verifikation der Automatisierungssoftware wurde akut um 2000 bearbeitet mit Konzentration auf *Safety-* und *Liveness Properties*. [Bri+02] verwendet den SPIN Modelchecker [Hol97], [Bau+04b] UPAAL oder [Sme+00] einen SMV-Abkömmling. [LC+99; YF03] fassen die verschiedenen Ansätze zusammen. IC3 wurde vorgestellt in [McM03] und weiter verbessert in [Bra11]. [Fel+14] ist ein ähnlicher Ansatz für die Regression Verification von Integerprogrammen mit dem Versuch, IC3 Klauseln zu ermittlen, die die Gleichheit belegen. [Bau+04a; Bor+00] liefert eine Formalisierung für SFC.

Wissenschaftlicher Beitrag

Dies ist die erste veröffentlichte Arbeit über die Regression Verification von Automatisierungssoftware. Wir stellen Software für die Transformation von ST und SFC über ST_0 zu SMV Modellen zur Verfügung. Daneben definieren wir die bedingte Äquivalenz mit Überlegungen für Anforderungen an den Äquivalenzbeweisen, um die Soundness zu erhalten.

Ausblick

Diese Arbeit stellt nicht den Abschluss der Überlegungen dar. Offene Punkte sind die Erweiterung auf die anderen drei Programmiersprachen im IEC 61131-3, die Optimierung der generierten Modelle und des gesamten Beweisprozesses wie zum Beispiel die Zerlegung in kleine Teilbeweise.

Herausfordernd dürfte sich die Zugänglichmachung der Verifikation für die Ingenieure gestalten. Mit den neuen Operatoren haben wir den Versuch angetreten, wiederverwendbare und verständliche Hilfsmittel zur Modellierung der Bedingung zu geben. Eine andere Richtung ist die Herleitung der Bedingung aus den syntaktischen Unterschieden zwischen zwei Softwareversionen. Hier wäre ein Ansatz über die Aufnahme von Gegenbeispiel aus dem Modelchecker in die Bedingung als machbar anzusehen. Ähnlich verfährt IC3 bei der Suche nach den induktiven Klauseln. Eine solche Bedingung muss mit der Erwartung des Ingenieurs bzw. mit den Anforderungsänderungen überprüft werden.

Contents

1. Introduction

Motivation

Plants are a long-term investment and run over many years, or even decades. Market pressure and technical progression require changes in the hardware and software of the plant. The plant evolves and adapts to new situations. The operator replaces hardware components, and updates the software to ensure that the plant is operational in the new composition. A desirable aim is the preservation of the old plant behaviour after the evolution. But the evolution has an intention, for example efficiency, fixing defects, reducing abrasion or the support of new workpieces. Taking into account, plants are a large investments with claims on safety for the staff. This question becomes acute: *How can we ensure, that the purposed behaviour of the plant processes stays the same, and how can we detect the introduction of errors and violation of plant safety?*

Regression testing is a technique for finding the introduction of bugs. For this purpose the test engineers creates a suite of test cases, which both software revisions have to pass. A test case consist of an expected result and input instance for the tested functionality. If we use the output of the previous revision as the reference result, we test for equivalence to the predecessor. But the equivalence only covers the input instances from the test cases. This disadvantage tackles the regression verification. Regression verification tries to prove the equality of two revisions by usage of verification techniques. The prove covers all possible instances and the achieved equivalence statement is stronger. But the reference for correctness is previous version, hence regression verification shows that the new revision is correct or defect like predecessor.

Equivalence between plants is a soft notion. We have to sharpen the term in several ways. First, the plant has different layers, there the components are located. The equivalence can talk about the single layers: physical measurement, the sensor or actuator values, just the variables within the automation software, or bundle them into a new system. Second, the changes between revisions have an intention, for example fixing bugs or supporting different hardware. If we show the equivalence of both revision, regardless of the layer, we need to exclude the behaviour, which was inflicted by intentional changes. The remaining behaviour should be the common base of both revisions and behave equally. Third, a translation of the output and input values between both revision are necessary if the environment with his sensor and actuators has changed. So the equivalence underlies an abstraction level, conditions and translations. We give in Section 1.1 an example for a plant.

Automation

A plant consists of hardware pieces, that are controlled by the Programming Logic Controller (PLC). The PLC is the central controlling unit, that cyclic executes the automation

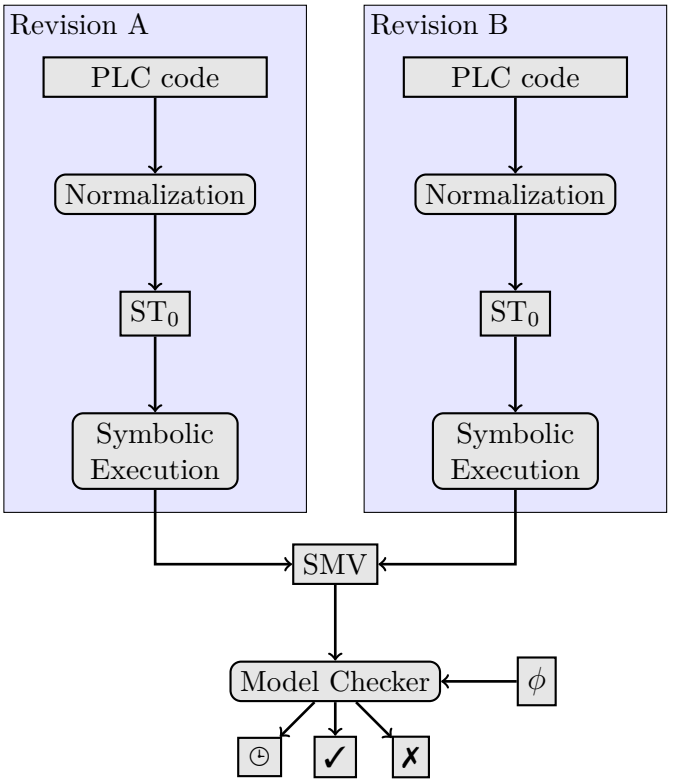


Figure 1.1 Overview over the verification process. PLC code from both versions are normalized into a subset of ST called ST_0. With symbolic execution we create modules in SMV, that are tested for equivalence in the model checker. ϕ describes the condition for the equivalence. The model checker decides whether the equivalence holds or not, respective it hits our personal timeout.

software. The sensors and actuator values are transferred over a bus system. The IEC 61131 norm [Com02b] standardize the automation of plants with on hardware and communication as long with five textual and graphical programming languages. The combination of software and hardware implies an particular awareness of safety requirements. Such a plant can harm humans, for example every plant requires an emergency stop.

Focus of Work

In this thesis we use verification techniques for proving the equivalence of PLC software. This excludes the reading and writing on the bus system, equivalence on observable behaviour of the plant or the equivalence non-functional requirements. We do not check individually for safety property, instead if the behaviour from the previous revision is safe, then the new revision is safe, under the assumption of the equivalence. Additionally, we assume that the PLC is crucial decider of the plant behaviour, and the sent PLC commands causes the same actions and effects in the plant. As conclusion, a change of an equivalent PLC does not change the plant behaviour. We cover the cycle operation mode of PLC, in which the software is triggered in fix time intervals. Additionally, we focus on the programming languages Structured Text (ST) and Sequential Function Chart (SFC), without loss of generality, because the other languages are reducible to ST. Figure 1.1 gives the big picture of the processing pipeline.

Both software revisions are translated into Symbolic Model Verifier (SMV) format. The nuXmv[1] [Cav+14] model checker proves with IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness) [Bra11; McM03] our equivalence invariants between translated software revisions.

We use the *Pick-and-Place Unit* case study [VH+14] from the Technical University Munich

[1] `http://nuxmv.fbk.eu`

for evaluation. The case study consists of 15 software revisions for a plant, with changes in soft- and hardware. We treat every revision, which introduces changes to software, by proving the equivalence to the predecessor revision.

Outline

We begin with the introduction of our continuous example (Section 1.1) for this thesis. The example is reused in the following chapter to illustrate the presented methods. The Figure 1.1 gives overview about verfication process and about this master thesis. We start with the PLC code in Chapter 2 with the introduction of Structured Text (Section 2.2) and Sequential Function Chart (Section 2.3). We formalize both languages with their syntax and semantics. Our aim is transfer both languages into an intermediate representation ST_0 (Section 2.4) for the translation into SMV models (Section 3.3). In Section 3.1 we make theoretical observations with PLC as the controller of a cyber-physical system. For the regression verification we introduce an extensive definition of *conditional* program equivalence (Section 3.2) and decode several derivation of conditional equivalence in SMV (Section 3.4). In Section 3.5 we formalize our observation of finding the conditions between the case study scenarios to an heuristic approach. Chapter 4 explains the case study, the difference between the revisions and presents the equivalence conditions and performance results. In Chapter 5 we discuss open and further aspects.

Contributions

This is the first work on the regression verification of PLC software. We provide a software for translating Structured Text and Sequential Function Chart source code into SMV modules, define a new equivalence notion: *conditional equivalence*, lay the foundation of the regression verification between plants and show feasibility on the basis of sophisticated software for real academic plant.

1.1 Introductory example

We introduce an motivational example of this work. Later we will pick it up in Section 2.5, Section 3.3 and Section 3.5 for an illustration of the presented steps.

Figure 1.2 shows a conveyor belt with a crane at the right end. An human operator puts a workpiece at the left end. The workpiece should be transported to the right and finally picked up by the crane. In this scenario the workpiece disappears after the pick up by the crane. We have light barrier sensors for detecting the presence of workpieces at the left and right belt end ($w_{1,2}$ in Figure 1.2). A boolean variable *run* controls the movement of the conveyor belt. If *run* is true, the top plane of the conveyor belt moves to the right. The crane is autonomous and offers one atomic operation *pickup* for grabbing a workpiece.

The plant should have following behaviour. If a workpiece is presence at the left edge, the conveyor starts moving, so that the workpiece is transported to right. If the workpiece reaches the position of the crane on the right edge, the conveyor stops and crane picks up the workpiece. After pickup, the plant is ready to process the next workpiece. Additionally to this behaviour, an human operator can abort this process with an emergency stop. In the case of an emergency stop the conveyor and the crane halts. The emergency stop resets the plant to the starting state, waiting for a workpiece at the left end.

Our example serves like a base plant for different kinds of extensions. For example we could enhance the plant with sorting, arrangement or different processing capabilities for workpieces. The next revision (Figure 1.3) introduces a detector, denoted by D, for distinguishing between intact and broken workpieces. We want to reject the broken workpieces from the intact ones, by picking up the intact ones with the crane. The broken workpieces end up in the rubbish bin at the right side.

The transportation and the pickup by the crane is the common part of both revisions. We observe an equivalence between both revision, if only intact workpieces appears on the conveyor belt.

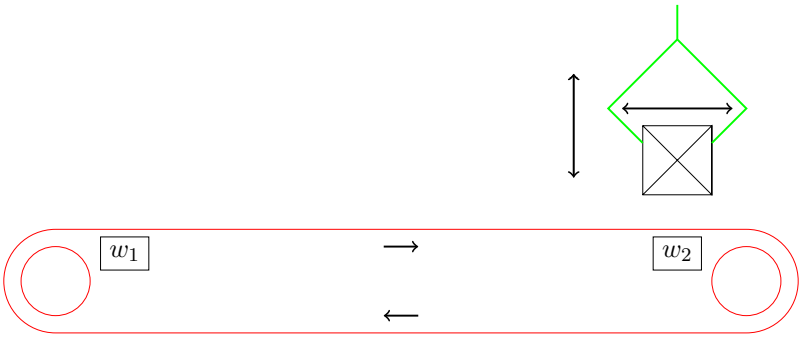


Figure 1.2 First revision of the introductory example. This plant can just move workpiece from left to right and pick them up by the crane. It has two light barrier sensors $w_{1,2}$ for detecting the presence of a workpiece, a crane and a conveyor belt.

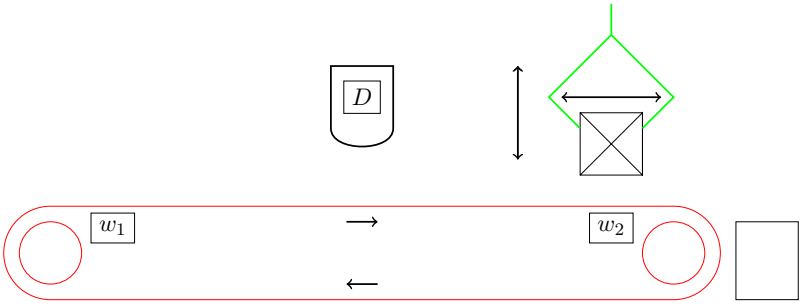


Figure 1.3 Second revision of the introductory example. The hardware components the same like Figure 1.2, except the detector D and the rubbish bin at the left. This plant sorts between intact and broken workpiece.

Proposition 1.1 (Equivalence of Introductory Example) *Revision in Figure 1.3 behaves the same like revision from Figure 1.2 if and only if intact workpieces appears.*

The Proposition 1.1 let many points open. The equivalence of plants can be described in many ways. We could simulate both revisions with same workpieces and observe if the conveyor belt and crane do the same operations. In this thesis we concentrate at the input and output values of the PLC in both revisions and assume that plant behaves equivalent, if the PLC software is equivalent. We compare the output results of the PLC in every turn between both revisions, under the same input. Before we can prove Proposition 1.1 we need to write the PLC software (Section 2.5) and translate the software into SMV (Section 3.3.1).

1.2 Related Work

We conjoin two disciplines of software verification, verification of automation programs and the regression verification.

Automation Verification

Around 2000 the verification of safety properties in automation software was a popular topic. This era is covered in [YF03] with an overview and classification of covered IEC 61131 language and verification approach of various papers. Additionally the survey [LC+99] gives a more detail overview of transformation processes for program languages to verifiable models. [LC+99] shows two different approaches for verification of SFCs: theorem proving and model-checking approach. Both approaches are developed on the SFC's ancestor standard Grafcet [Com02a]. The authors state validity for SFC. In theorem proving the SFC is translated into theorems and proved in combination with axioms for SFC. For model-checking approach the SFC is translated into transition system for various kind

of models, for example timed automata, hybrid automata or state machines. All papers are dedicated to checking safety or liveness properties and limited to subset of IEC 61131 languages or language features. Especially the ST and SFC dialect are of interest for us. We narrow the discussion to these dialects. [Sme+00] covers all languages of IEC 61131 by translating them into transition diagrams. A scan cycle consists of multiple transition steps. In our approach every scan cycle execution becomes one transition in the model checker. [Sme+00] leads to bigger transition diagrams with smaller state changes. Accordingly the reached states at the end of a scan cycle has to marked specially. Only in the marked states the safety properties must hold. [Bor+00] present a method for SFC verification. They translate the SFC directly into SMV modules for the model checker, with support for nested SFCs, but the SFC actions contain only assignments. [Bor+00] paper gives a first formalization of SFC, that is extended in [Bau+04a] with covering of timed SFC. [Bau+04a] is a companion paper to [Bau+04b], in which the authors illustrate verification for safety properties of untimed SFC with a SMV model checker and timed SFC with the Uppaal model checker. A completely different approach is in [Bri+02] with SPIN [Hol97]. [Bri+02] build a model to check real-time properties among others with an abstraction of time for a example plant.

Regression Verification

Regression Verification has a more general focus. The discipline tries to make equivalence statements sophisticated program with recurrence and loops. [GS09] describes regression verification as the "prove that it [the software] is 'as correct' as the previous version". Godlin and Strichman have published many papers on regression verification [GS09; GS13; Str09]. They engage in regression verification on C programs with respect to loops and recursion. In an earlier work [GS08] from 2008 they state different kinds of equivalence:

1. **Partial equivalence:** Given the same inputs, any two terminating executions of P_1 and P_2 return the same value.
2. **Mutual termination:** Given the same inputs, P_1 terminates if and only if P_2 terminates.
3. **Reactive equivalence:** Given the same inputs, P_1 and P_2 emit the same output sequence.
4. **k-equivalence:** Given the same inputs, every two executions of P_1 and P_2 where
   - each loop iterates up to k times, and
   - each recursive call is not deeper than k,

   generate the same output.
5. **Total equivalence** The two programs are partially equivalent and both terminate.
6. **Full equivalence:** The two programs are partially equivalent and mutually terminate.

We categorize our equivalence approach as *conditional equivalence*. We claim the equivalence of both revision only if the given condition holds. The condition fades out the equivalence in cases with intentional changed behaviour (Section 3.2). In the category of [GS08] we categorize the equivalence of PLC software as reactive. Divergent run is not allowed in the scan cycle setup (Chapter 2). [Fel+14] gives an automatic approach for determining coupling invariant for supporting an equivalence claim, by solving derived Horn constraints from the two given programs. In the background of the constraint solving algorithm works IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness). IC3 is introduce in [McM03] and refined in [Bra11]. From the hardware verification comes the term of Sequential Equivalence Check (SEC). SEC describes the equivalence of sequential circuits, also circuits with an internal memory, like the PLC.

Sequential Equivalence Check

2. Formalisation of PLC software

Overview

In this chapter we define the function *eval*, that describes evaluation of Structured Text (Section 2.2) and Sequential Function Chart (Section 2.3) code. *eval* describes the operational semantics, also how an execution of language fragments changes the memory of the PLC.

With the *eval* definition for SFCs, we are able to give a reduction of SFCs with simultaneous constructs into an equivalent SFC without these constructs (Section 2.4.1). Our goal is an intermediate representation in ST_0 for the base of the verification (Section 2.4). ST_0 is the language, on which we define later the translation into SMV model (Section 3.3). We align the explanation of the language structure with the case study [VH+14]. So we leave out parts of the standard, that are not present in the case study. Especially we leave out the configuration and tasks definition. Both are defined in IEC 61131-3, but often the implementation is vendor and device specific.

We start the syntax and semantics of PLC software after IEC 61131-3. We use [TJ09] and [Neu+00] as the reference to the language. The IEC 61131-3 norm leave space for interpretation, so different behaviours between implementation of the standard are possible. Furthermore, the vendor augments the SFC or ST implementations with specific features. The case study is developed with CODESYS[1]. Defining a formalized execution model of the IEC 61131-3 languages, especially SFC, is research subject in various of papers [Bau+04a; Bor+00; Nar+10a; YF03].

Software Structure

We start with a top down view on the structure in Figure 2.1. The configuration is a composition of different tasks and contain shared variables or access path definitions. A task describes the scheduling of programs, for example event-based or cyclic execution in fixed time intervals. A program can be triggered by more than one task. If multiple tasks conflicts, the task priority determines the scheduling. We focus on cyclic invocations every n ms. The memories of different tasks are disjoint, but the configuration can allow the access to specific variables from other programs, under a local name. The syntax of configuration and tasks are often vendor specific, with the level of programs, the IEC 61131-3 programming languages take place. Programs are the entry point of execution like the *main* symbol in shared objects. Program's signature does not allow arguments or return values. The communication with the environment is configured on configuration level, as a binding between the program's variables and the bus system. Function blocks are instantiable functions and are allowed with programs or other function blocks. You can

[1] `http://www.codesys.com/products/codesys-engineering/development-system.html`

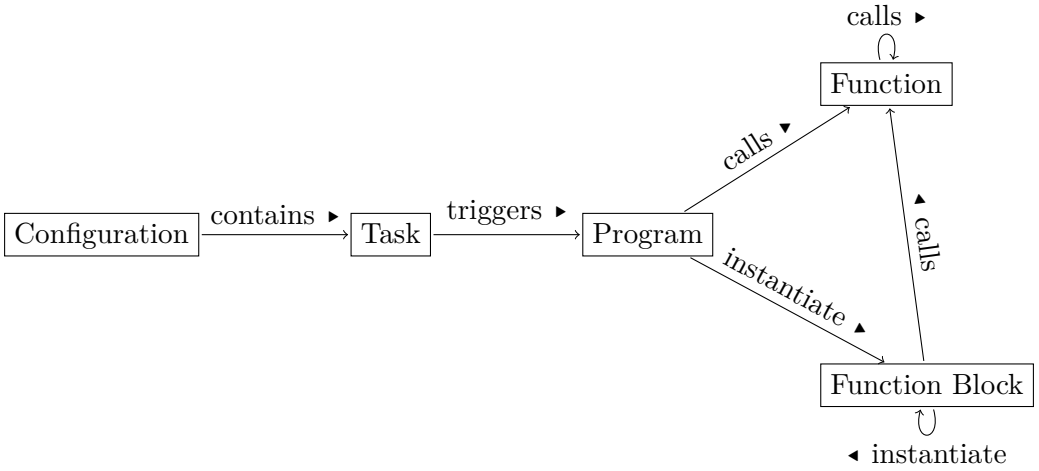


Figure 2.1 Structure of the programming language contstructs. One system configuration has multiple Tasks. A task triggers one program. Programs and function blocks can call and instantiate other function block. Only functions have a volatile state.

consider function blocks like classes with one function or like a function with a local state. Programs, function blocks and functions can call functions. Functions are pure in the sense, that they can not change the global state, and they the local state is volatile. They are not allowed to instantiate function blocks. A function block can not instantiate itself. Every memory is static and allocated before the program starts. An self instantiating would end in an infinite loop. The languages miss features for dynamical memory allocation or pointers. Our goal in regression verification is the equivalence of two PLC programs with the same scan cycle time.

Procedures

Programs, function blocks and function[2] are divided in two parts: the declaration and definition. The declaration defines all variables in this procedure scope, like in the programming language Pascal. We discuss the declaration in Section 2.1. The definition part holds the implementation in either one of the five languages:

- Structured Text (ST)
- Sequential Function Chart (SFC)
- Ladder Diagrams (LD)
- Function Block Diagram (FBD)
- Instruction List (IL)

LD, FBD and SFC are graphical representation. IL is an assembler language and ST is a high level with high similarities to the Pascal language family. The procedure implementation can be given in one of the five representations and is able to call every procedure independently of his implementation representation. In this thesis we focus on ST (Section 2.2) and SFC (Section 2.3), because they are constituent in the case study. IL, LD and FBD are not considered further. But the introduced technique should be valid for them, too. As far as we can see, they are transformable into ST and thereby transformable to ST_0 and SMV.

Execution Model

We develop a general concept about the execution of the five programming languages. Our goal is to cover the execution semantic into a function *eval* for ST and SFC. *eval* is extendable to IL, FBD and LD, if needed. In general, the execution of a program modifies the state of the system. In our case the system is an embedded computer, which stores the

[2]We define the notion “procedure” as notion for the three callable units.

state in a memory. We formalize the memory as function $\mathcal{V}$, that maps a variable identifier to a value and the function image can be changed for particular variables.

Definition 2.1 (Memory) *$\mathcal{V}$ represents the memory of the PLC, with*

$$\mathcal{V}\colon L(\langle name \rangle) \to \Delta, \tag{2.1}$$

where the domain of $\mathcal{V}$ are all possible variable identifiers, and the codomain Δ denotes every possible value.

Definition 2.2 (Memory Update) *Let $\mathcal{V}$ be a memory. We denote a update of the memory with brackets. An update of the variable x to value v leads to, that every access of x respond to v after the update:*

$$\mathcal{V}[x := v](x) =_{\text{def}} v \tag{2.2}$$

The complete definition is

$$\mathcal{V}[x := v](y) =_{\text{def}} \begin{cases} v & : y = x \\ V(y) & : else \end{cases} \tag{2.3}$$

Every variable has a value either by the declaration of an initial value or by the default of the data type. A look up of an undefined variable results into an error. $\mathcal{V}$ returns an undefined value $\boxdot$. Moreover, $\mathcal{V}$ handles his argument case insensitive, for instance $\mathcal{V}$(A and $\mathcal{V}$(a) denotes the same value.

We define the evaluation of a well-formed source code $\mathcal{R}$ in an arbitrary programming language, as the change of the memory.

Definition 2.3 (Evaluation) *eval executes the $\mathcal{R}$ in the memory $\mathcal{V}$ and returns the new memory assignment $\mathcal{V}'$*

$$eval\colon \mathcal{V} \times \mathcal{R} \to \mathcal{V}, \tag{2.4}$$

where $\mathcal{V}$ is the set of all possible memories and $\mathcal{R}$ denotes of all possible programs.

Grammar Notion

In this thesis we describe the syntax for the languages as context-free grammars in similar form as EBNF. The atomic statements are the literals, marked with single quotes, and non-terminals, denoted by angle brackets. Consecutively statements forms a sequence. A star a^* denotes, that the expression a can be repeated arbitrary often, including zero. If we exclude the zero-repetition, we write a^+. An optional statement a is written as $a?$. The bars marks exclusive choice of separated cases. The paren is only for making the precedence clear.

2.1 Variables and Data types

A procedure declaration (2.4) consists of multiple variable declarations $\langle decl \rangle$. A variable declaration begins with an variable keyword and can define multiple variables with identifier, data type and initial value.

Identifier

The variable identifier is case insensitive, hence “a” and “A” denotes to the same memory location. Moreover, device dependent, there is a fix amount of significant digits, to which

```
1  FUNCTION_BLOCK A
2  VAR b : B;
3  {* ... *}
4  END_FUNCTION_BLOCK
```

```
1  FUNCTION_BLOCK B
2  VAR a : A;
3  {* ... *}
4  END_FUNCTION_BLOCK
```

Figure 2.2 Example of cyclic variable instantiation

the variable identifier is truncated. So "abc1" and "abc2" links to the same memory, if the significant digits count is less than four. After [TJ09] current PLCs systems have 32 significant digits. The identifier follows the usual rules: start with a alphabetical letter or underscore, consecutively letters may be alphanumerical or underscore.

Variable Keyword The variable keyword defines the reading and writing permissions from inside (callee) and outside (caller) of the procedure. Input variables are readable and not writable within the procedure, but writable and not readable from the outside, vice versa for output variables. Normal variables are readable and writable from the inside and not accessible from the outside. The caller and callee can write and read from in-out variables. It is possible to declare a variables as global ('VAR_GLOBAL') and access these variables from other procedure ('VAR_EXTERNAL'). We omit these variables in this thesis. They map between input and output channels on the bus to PLC internal variables. Access variables are defined on configuration level.

Variable Attributes A variable can have attributes. In our case 'CONSTANT' is the only relevant one. 'RETAIN' and 'NON_RETAIN' describes the behaviour on restart. Retained variables are cached in non-volatile memory and restored on a warm restart. In all other cases the variables are initialized accordingly to the data types initial value (Table 2.1). 'R_EDGE' and 'F_EDGE' modifies boolean variables only. Such boolean values are presented as a raising or falling edge on the bus.

$\langle name \rangle$ denotes a valid identifier name. $\langle type \rangle$ is the name of a built-in data type, a user-defined data type or the name of function block. The $\langle literal \rangle$ denotes the literal of an initial value.

Definition 2.4 (Declaration grammar)

$$\langle decls \rangle \rightarrow \quad (\langle decl \rangle)^* \tag{2.5}$$

$$\begin{aligned} \langle decl \rangle \rightarrow \quad & (\text{'VAR'} \mid \text{'VAR\_INPUT'} \mid \text{'VAR\_OUTPUT'} \mid \text{'VAR\_IN\_OUT'} \\ & \text{'VAR\_GLOBAL'} \mid \text{'VAR\_EXTERNAL'}) \\ & (\langle attribute \rangle)? \\ & (\langle name \rangle \text{ ':' } \langle type \rangle \text{ (':=' } \langle literal \rangle \text{)? ';')}^+ \\ & \text{'END\_VAR'} \end{aligned} \tag{2.6}$$

There is no concept of dynamical allocation or deallocation of variables. Every variable is allocated at the start the PLC hardware. It is not possible to have cyclic variables instantiation (Figure 2.2), for instance a function block A instantiate a function block B and B instantiate B.

Data types The overview of the data types is in Table 2.1. These are the built-in data types, that can be derived or composed into new data types. The runtime promotes variables to the next biggest type, for example the addition from SINT and LINT results into a LINT. Arithmetic overflows can occur. The standard provides functions for explicit casting between the data

Name	Size	Initial	Range
ANY_BIT			
BOOL	1	0	
BYTE	8	0	
WORD	16	0	
DWORD	32	0	
LWORD	64	0	
ANY_INT			
SINT	8	0	$[-128,\ 127]$
INT	16	0	$[-32768,\ 32767]$
DINT	32	0	$[-2^{31},\ 2^{31}-1]$
LINT	64	0	$[-2^{63},\ 2^{63}-1]$
ANY_INT			
USINT	8	0	$[0,\ 255]$
UINT	16	0	$[0,\ 65535]$
UDINT	32	0	$[0,\ 2^{32}-1]$
ULINT	64	0	$[0,\ 2^{64}-1]$
ANY_REAL			
REAL	32	0.0	IEEE 754
LREAL	64	0.0	IEEE 754
ANY_DATE			
DATE			`D#0001-01-01`
TIME_OF_DAY			`TOD#00:00:00`
DATE_AND_TIME			`DT#0001-01-01-00:00:00`
TIME			`T#0`
STRING	8/char		`''`
WSTRING	16/char		`""`

Table 2.1 Overview about the built-in data types with ranges after [TJ09].

types. Literals have a prefix, which defines the data type. For some data types, for instance time or date, the prefix is mandatory, for other types the prefix is optional.

User-defined data types

User-defined data types are introduced with the 'TYPE' keyword. They are ether a derivation of a data type, or a declaration of a new one. A derivation can limit the range or set a different initial value of a data type. A declaration creates either a enumeration, array or a structure.

```
1  TYPE
2    Speed   : UINT (10..100);
3    Vector  : ARRAY[0..2] OF LREAL := [ 1.2, 5.2, 6.2 ];
4    Crane_T :
5      STRUCT
6        postion  : Vector;
7        pressure : UINT := UINT#5;
8        open     : BOOL := BIT#1;
9      END_STRUCT;
10 END_TYPE
```

Figure 2.3 Declaration of types

Definition 2.5 (Grammar of Type Declaration)

$$\langle tdecl \rangle \rightarrow \text{'TYPE'} \quad (2.7)$$
$$(\langle name \rangle \text{ ':' } \langle type \rangle \text{ ';'})^+$$
$$\text{'END\_TYPE'}$$
$$\langle type \rangle \rightarrow \langle name \rangle \ (\langle trange \rangle)? \ (\text{':='} \ \langle literal \rangle)? \quad (2.8)$$
$$| \ \langle tstruct \rangle$$
$$| \ (\langle tarray \rangle \mid \langle tenum \rangle) \ (\text{':='} \ \langle literal \rangle)?$$
$$\langle trange \rangle \rightarrow \text{'('} \ \langle literal \rangle \text{ '..' } \langle literal \rangle \text{ ')'} \quad (2.9)$$
$$\langle tenum \rangle \rightarrow \text{'('} \ \langle name \rangle \ (\text{','} \ \langle name \rangle)^* \text{ ')'} \quad (2.10)$$
$$\langle tstruct \rangle \rightarrow \text{'STRUCT'} \quad (2.11)$$
$$(\langle name \rangle \text{ ':' } \langle type \rangle \text{ ';'})^+$$
$$\text{'END\_STRUCT'}$$
$$\langle tarray \rangle \rightarrow \text{'ARRAY ['} \ \langle uint \rangle \text{ '..' } \langle uint \rangle \ (\ \text{','} \ \langle uint \rangle \text{ '..' } \langle uint \rangle \ )^* \text{ ']'} \quad (2.12)$$
$$\text{'OF'} \ \langle name \rangle$$

Structs can be nested, but have to be cycle free, else the pre-allocation of the data structures can not be made at the hardware start. Figure 2.3 gives an example of the defined grammar, where `Speed` is a derived UINT in the inclusive interval from 10 to 100, `Vector` an array of LREAL and `Crane_T` a structure with three elements. The complex built-in types in Table 2.1 can be implemented as structures. Nevertheless, for efficiency date and time data type are encoded as integer.

2.2 Structured Text

ST is the only high-level textual programming language in IEC 61131-3. It is similar to the Pascal language. In this section we define the syntax and semantics of expression and statements in ST. We give the syntax in the defined form from the beginning of this chapter and semantic in form of the eval function. Syntax and semantics are reusable for ST_0.

For the definition of the ST semantics we need a concept of execution configuration. The configuration consists a memory $\mathcal{V}$ (Definition 2.1) and the *rest* program. The rest program is the part of the source code, that needed to be executed. It is similar to maintaining a program counter, that points to the current source code line.

Definition 2.6 (Execution Configuration) *The execution configuration is a tuple* $(\mathcal{V}, \mathcal{R})$*, where* $\mathcal{V}$ *is the current memory and* $\mathcal{R}$ *is the current position in the program code, decoded as the rest program, that needed to be executed.*

As an invariant, the rest program is always a well-formed ST source code and from the set R_{ST} of all ST programs.

We split the explanation of ST into two parts. First, we look on the evaluation of expression, later we define the statements. The expression grammar follows the rules for Pascal. A variable name (2.17) or a literal (2.18) are the base cases. Arbitrary expressions can be combined with a binary operator (2.13), unary operator (2.14), in a function call (2.15) or just wrapped in paren (2.16).

Expression Grammar

Definition 2.7 (Grammar of Expression)

$$
\begin{aligned}
\langle expr \rangle \rightarrow\ & \langle expr \rangle \langle binop \rangle \langle expr \rangle && (2.13)\\
|\ & \langle unary \rangle \langle expr \rangle && (2.14)\\
|\ & \langle name \rangle \text{ '(' } \langle expr \rangle \text{ (',' } \langle expr \rangle \text{)* ')'} && (2.15)\\
|\ & \text{'(' } \langle expr \rangle \text{ ')'} && (2.16)\\
|\ & \langle name \rangle && (2.17)\\
|\ & \langle literal \rangle && (2.18)\\
\langle binop \rangle \rightarrow\ & \text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'**'} \mid \text{'MOD'} \mid \text{'/'} \mid && (2.19)\\
& \text{'<'} \mid \text{'>'} \mid \text{'<='} \mid \text{'>='} \mid \text{'='} \mid \text{'<>'} \mid\\
& \text{'\&'} \mid \text{'AND'} \mid \text{'OR'}\\
\langle unaryop \rangle \rightarrow\ & \text{'NOT'} \mid \text{'-'} && (2.20)
\end{aligned}
$$

The definition does not care about left recursion or ambiguity. We define the operator precedence separately in descending binding strength.

Definition 2.8 (Operator Precedence) *The precedence of the operator is*

1. *parenthesis* (2.16)
2. *function call* (2.15)
3. *unary arithmetic and logical negation* (2.14)
4. *power ('**')*
5. *point arithmetic ('*', '/', 'MOD')*
6. *line arithmetic ('+' and '-')*
7. *numerical comparison ('<', '>', '<=', '>=')*
8. *equality ('=') and non-equality ('<>')*
9. *boolean operators ('AND', 'XOR', 'OR').*

We define the evaluation of an expression by induction. Beginning with the base cases with variable names and literals. The induction steps works with the different operators.

Expression Interpretation

Definition 2.9 (Semantics of expressions) *We define the interpretation $I_{\mathcal{V}}$ of expression $e \in L(\langle expr \rangle)$ in a given memory $\mathcal{V}$*

$$I_{\mathcal{V}} \colon L(\langle expr \rangle) \rightarrow \Delta, \tag{2.21}$$

where Δ denotes the set of all possible values.

base *Let the given expression e be a literal or name of a variable, then we evaluate the value directly or lookup in the current configuration.*

$$I_{\mathcal{V}}(e) =_{\text{def}} \begin{cases} \mathcal{V}(e) & : e \text{ is a name} \\ c_e & : e \text{ is a literal} \end{cases} \tag{2.22}$$

$c_e \in \Delta$ denotes the value for the corresponding literal e.

induction *In the following we differ between the identifier f of a function or $\otimes$ for an operator and $\otimes_M$ the concrete implementation on the data level. The implementation is denoted with an index M. We distinguish between the cases of $\langle expr \rangle$.*

binary operation *Let $e = g \otimes h$ be an expression of a binary operator $\otimes$, like '+' or '**', the g, h are expressions. We interpret as recursively*

$$I_{\mathcal{V}}(g \otimes h) =_{\text{def}} \otimes_M(I_{\mathcal{V}}(g),\ I_{\mathcal{V}}(h)). \tag{2.23}$$

unary operation *Let $e = \otimes g$ be an unary expression, then $\otimes$ is an unary operation and g an arbitrary expression. We interpret as follows:*

$$I_{\mathcal{V}}(\otimes g) =_{\text{def}} \otimes_M(I_{\mathcal{V}}(g)) \tag{2.24}$$

function call *Let $e = f(g_1, \ldots, g_n)$ be a function call of f with arguments $g_1, \ldots, g_n$. First, we interpret the arguments, then we apply the corresponding function f_M.*

$$I_{\mathcal{V}}(f(g_1, \ldots, g_n)) =_{\text{def}} f_M(I_{\mathcal{V}}(g_1), \ldots, I_{\mathcal{V}}(g_n)) \tag{2.25}$$

Expression are side-effect free, hence $I_{\mathcal{V}}$ does not modify the memory $\mathcal{V}$. Please note, that it is not allowed to call a function block within an expression. Only application of functions are allowed. A function block does not have a declared return value and type. Instead you have to access the output variables separately. It is possible to sum up the induction step into the case of function calls, with predefined function for the built-in unary and binary operators. Please note, f_M is either a built-in function or defined by the user. In the second case requires an appropriate *eval* function for the implementation of f. Next, we will define such a function $eval_{ST}$ for ST, beginning with the syntax.

Statement Grammar

ST has following statements: assignment (2.28), function block call (2.29), conditionals statements if (2.31) and case (2.32) and three loops for (2.34), while (2.35) and repeat (2.36). Keyword statements exit and returns manipulate the control flow. EXIT terminates a loop premature, respective RETURN a procedure. Examples of ST code are in Figure 2.16 and figures A.1 to A.3. We give an easier version of the grammar of [Nar+10b].

Definition 2.10 (Grammar of Statements)

$$\langle los\rangle \rightarrow \langle statement\rangle \text{ (';' } \langle statment\rangle)^* \tag{2.26}$$

$$\begin{aligned}\langle statement\rangle \rightarrow\ & \langle assign\rangle \mid \langle call\rangle \mid \langle if\rangle \mid \langle case\rangle \\ & \mid \langle for\rangle \mid \langle while\rangle \mid \langle repeat\rangle \\ & \mid \text{'RETURN'} \mid \text{'EXIT'}\end{aligned} \tag{2.27}$$

$$\langle assign\rangle \rightarrow \langle name\rangle \text{ ':=' } \langle expr\rangle \tag{2.28}$$

$$\langle call\rangle \rightarrow \langle name\rangle \text{ '(' } \langle param\rangle \text{ ')'} \tag{2.29}$$

$$\begin{aligned}\langle param\rangle \rightarrow\ & \langle name\rangle \text{ ':=' } \langle expr\rangle \\ & \mid \langle name\rangle \text{ '=>' } \langle name\rangle \\ & \mid \langle param\rangle \text{ (',' } \langle param\rangle)?\end{aligned} \tag{2.30}$$

$$\begin{aligned}\langle if\rangle \rightarrow\ & \text{'IF' } \langle expr\rangle \text{ 'THEN' } \langle los\rangle \\ & (\text{'ELSEIF' } \langle expr\rangle \text{ 'THEN' } \langle los\rangle)^* \\ & (\text{'ELSE' } \langle los\rangle)? \\ & \text{'END\_IF'}\end{aligned} \tag{2.31}$$

$$\begin{aligned}\langle case\rangle \rightarrow\ & \text{'CASE' } \langle expr\rangle \text{ 'OF' } (\langle cexpr\rangle \text{ ':' } \langle los\rangle)^+ \\ & (\text{'ELSE' } \langle los\rangle)? \text{ 'END\_CASE'}\end{aligned} \tag{2.32}$$

$$\begin{aligned}\langle cexpr\rangle \rightarrow\ & \langle integer\rangle \mid \langle name\rangle \mid \langle integer\rangle \text{ '..' } \langle integer\rangle \mid \\ & \langle cexpr\rangle \text{ (',' } \langle cexpr\rangle)?\end{aligned} \tag{2.33}$$

$$\begin{aligned}\langle for\rangle \rightarrow\ & \text{'FOR' } \langle identifier\rangle \text{ ':=' } \langle expr\rangle \text{ 'TO' } \langle expr\rangle \text{ ('BY' } \langle expr\rangle)? \\ & \text{'DO' } \langle los\rangle \text{ 'END\_FOR'}\end{aligned} \tag{2.34}$$

$$\langle while\rangle \rightarrow \text{'WHILE' } \langle expr\rangle \text{ 'DO' } \langle los\rangle \text{ 'END\_WHILE'} \tag{2.35}$$

$$\langle repeat\rangle \rightarrow \text{'REPEAT' } \langle los\rangle \text{ 'UNTIL' } \langle expr\rangle \text{ 'END\_REPEAT'} \tag{2.36}$$

We need the operational semantics for manipulating the memory $\mathcal{V}$ with ST statements. We have already defined the configuration $\mathcal{C}$. Restating Definition 2.6, a configuration contains the current memory $\mathcal{V}$ and the ST rest program $\mathcal{R} \in L(\langle los\rangle)$. In Definition 2.11 we transform a configuration into a successor configuration by executing the first statement $\mathcal{R}_0$ of the rest program $\mathcal{R}$ We note the relation between current and next configuration as follows

$$\frac{\mathcal{V} \quad \| \quad \mathcal{R}_0;\mathcal{R}'}{\mathcal{V}' \quad \| \quad \mathcal{R}'} \tag{2.37}$$

where the current configuration $\mathcal{C} = (\mathcal{V}, \mathcal{R}_0;\mathcal{R}')$ and the next configuration $\mathcal{C}' = (\mathcal{V}', \mathcal{R}')$. We assume at the end every procedure an implicit return statement. So the execution terminates if the rest program $\mathcal{R}$ is the return statement.

Definition 2.11 (Semantics of statements) *Let $\mathcal{C} = (\mathcal{V}, \mathcal{R})$ be a the current execution configuration. We divide $\mathcal{R} = \mathcal{R}_0 R'$ in the current statement $\mathcal{R}_0$ and the rest $\mathcal{R}'$.*

assign *An assignment modifies the memory at a certain position, given by the variable name.*

$$\frac{\mathcal{V} \quad \| \quad \mathbf{a := e}\ ;\mathcal{R}'}{\mathcal{V}[a := I_{\mathcal{V}}(e)] \quad \| \quad \mathcal{R}'} \tag{2.38}$$

return *If $\mathcal{R}_0$ is the return statement, we halt the execution.*

$$\frac{\mathcal{V} \quad \| \quad \mathbf{RETURN} \; ; \mathcal{R}'}{\mathcal{V} \quad \| \quad \boxdot} \tag{2.39}$$

call *If $\mathcal{R}_0$ is a function block call, we need to evaluate the arguments, call the eval function on the procedure body, and transfer the output variables into the caller memory. Moreover the function block have a local state, we assume that this state is stored within the caller memory under the instance name. The startup initialization of the PLC assures, that such a memory exists for every initialized function block.*

We apply following steps.

1. *We need to look up the procedure body of f and obtain an arbitrary IEC procedure body. Let R_f the procedure body.*

   Additionally we obtain a the internal memory of the function block $\mathcal{V}_f$ from the caller memory.

$$\mathcal{V}_f := \mathcal{V}(f) \quad . \tag{2.40}$$

2. *Update all given input slots a_i with $a_i := e_i$ in the new configuration:*

$$\mathcal{V}'_f(s_i) := I_{\mathcal{V}}(e_i) \tag{2.41}$$

3. *Execute the function block and store the new state*

$$\mathcal{V}''_f := eval(\mathcal{V}'_f, \mathcal{R}_f). \tag{2.42}$$

4. *We update the internal memory in the caller's memory:*

$$\mathcal{V}' =_{\text{def}} \mathcal{V}[f := \mathcal{V}''_f]. \tag{2.43}$$

   Additionally set every output parameter $o_1 \Rightarrow t_1, \ldots, o_n \Rightarrow t_n$ into the caller memory:

$$\mathcal{V}'' =_{\text{def}} \mathcal{V}'[t_i := I_{\mathcal{V}'_f}(o_i), \; \ldots \; , t_n := I_{\mathcal{V}'_f}(o_n)] \tag{2.44}$$

The new configuration is $\mathcal{C}' =_{\text{def}} (\mathcal{V}'', S_1)$.

In short

$$\frac{\mathcal{V} \| f(a_1 := e_1, \ldots, a_n := e_n, o_1 \Rightarrow t_1, \ldots, o_m \Rightarrow t_m); \mathcal{R}'}{\mathcal{V}[f := eval(\mathcal{V}(f)[a_i := I_{\mathcal{V}}(e_i)], \mathcal{R}_f)][t_j := I_{V'}(o_j)] \| \mathcal{R}'} \quad , \tag{2.45}$$

where $a_i := I_{\mathcal{V}}(e_i)$ and $t_j := I_{\mathcal{V}}(o_j)$ stand for the update of every input or output variable with $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$.

if *Let $\mathcal{R}_0$ be an if statement. In the simple case, there are no alternative cases*

$$\frac{\mathcal{V} \parallel \textbf{IF } c \textbf{ THEN } \mathcal{R}_\mathcal{B} \textbf{ END\_IF } ;\mathcal{R}'}{\mathcal{V} \parallel \begin{cases} \mathcal{R}_B;\mathcal{R} & : I_\mathcal{V}(c) \text{ is true} \\ R & : I_\mathcal{V}(c) \text{ is false} \end{cases}} . \tag{2.46}$$

In general, a if statement has several branches with statements $\mathcal{R}_B^i$ protected by a condition c_i, in order of their occurrence in the if statement. The else branch is protected by $\top$ and we assume an empty statement lists if the else branch is omitted. We execute the statements $\mathcal{R}_B^i$, if i is the first branch where $I_\mathcal{V}(c_i) = \top$.

$$\frac{\mathcal{V} \parallel \textbf{IF } \ldots \textbf{ END\_IF } ;\mathcal{R}'}{\mathcal{V} \parallel \mathcal{R}_B^i;\mathcal{R}}, \tag{2.47}$$

where $i := \arg\min_{0 \leqslant i \leqslant n} \{I_\mathcal{V}(c_i)\}$.

case *We rewrite the condition of the cases of the case statement, so we fallback to an if statement. Let e be the expression of the case statement and c_i^j be the jth condition on the ith case entry.*

$$c_i =_{\text{def}} \bigvee_j g_e(c_i^j) \tag{2.48}$$

where g_e is function that handles the different types of case guards:

$$g_e(x) =_{\text{def}} \begin{cases} I_\mathcal{V}(e) = I_\mathcal{V}(x) & : x \text{ is an integer} \\ I_\mathcal{V}(e) = x & : x \text{ is an indentifier}^3 \\ a \leqslant I_\mathcal{V}(e) \leqslant b & : x \text{ is range expression with borders } a, b \end{cases} \tag{2.49}$$

The default case in $\mathcal{R}$ has the guard $\top$. Now, we have guards c_i corresponding statement blocks $\mathcal{R}_B^i$. The operational semantic is the same for if statements.

$$\frac{\mathcal{V} \parallel \textbf{CASE } e \textbf{ OF } \ldots \textbf{ END\_CASE } ;\mathcal{R}'}{\mathcal{V} \parallel \mathcal{R}_B^i;\mathcal{R}}, \tag{2.50}$$

where $i := \arg\min_{0 \leqslant i \leqslant n} \{I_\mathcal{V}(c_i)\}$.

while *Let $\mathcal{R}_0$ be a while loop, then the semantic is as follows:*

$$\frac{\mathcal{V} \parallel \textbf{WHILE } c \textbf{ DO } R_B \textbf{ END\_WHILE } ;\mathcal{R}'}{\mathcal{V} \parallel \textbf{IF } c \textbf{ THEN } \mathcal{R}_B;\mathcal{R}_0 \textbf{ END\_IF } ; \mathcal{R}'} \tag{2.51}$$

repeat *Let $\mathcal{R}_O$ be a repeat loop, then the semantic is as follows:*

$$\frac{\mathcal{V} \parallel \textbf{REPEAT } R_B \textbf{ UNTIL } c \textbf{ END\_REPEAT } ; \mathcal{R}'}{\mathcal{V} \parallel R_B; \textbf{ IF NOT } c \textbf{ THEN } \mathcal{R}_0 \textbf{END\_IF}; \mathcal{R}'} \tag{2.52}$$

for *We reduce for loops into while loops.*

$$\frac{\mathcal{V} \quad \| \quad \begin{array}{c}\textbf{FOR } i := start \textbf{ TO } stop \textbf{ BY } step \\ \textbf{DO } R_B \textbf{ END\_FOR } ; \mathcal{R}'\end{array}}{\mathcal{V}[i := I_{\mathcal{V}}(start)] \quad \| \quad \begin{array}{c}\textbf{WHILE } i < stop \\ \textbf{DO } R_B \ ; i := i + step \\ \textbf{END\_WHILE } ;\mathcal{R}'\end{array}} \tag{2.53}$$

Note, for loops does not guarantee a termination if a assignment in the loop body $\mathcal{R}_B$ overrides the loop variable i. The above operational semantic does not consider the exit statement.

Definition 2.12 ($eval_{ST}$) *The $eval_{ST}(\mathcal{V}, \mathcal{R}_{ST})$ function is defined over the operational semantic from Definition 2.11. The function $eval_{ST}$ returns the memory after termination $\mathcal{R} = \boxdot$.*

2.3 Sequential Function Chart

SFC boundary

SFC are based on the Grafcet standard [Com02a] and both inherit ideas from petri nets. A complete formalisation of all aspect are in [Bau+04a]. We state simpler model, sufficient for the case study. We do not specify timed behaviour or non-history SFCs. Hierarchical SFC, there SFC actions trigger other SFCs, are done over function blocks and is not part of the definition, like in [Bau+04a].

SFC components

A SFC is similar to finite state automata or petri net (Figure 2.4), but uses different terms. The states of a SFC are called steps and actions can be bound to them. Actions consist of a name and qualifier, which determines the moment, when they are triggered. We consider, *entry*, executed on entering the step, *exit*, executed on leaving the step, and *active*, executed during the step is active.

In literature they are often referred as P0, P1 and N. A transition connects steps via a guard. The guard can be expressed in an IEC 61131-3 language. Steps can be marked as active. An initial step is marked as active at the moment of initialization. Each active step holds a token, that is passed to the successor steps, iff. there is a transition and the transition guard is fulfilled.

fork-join split-merge

Divergences allow a case distinction, where the transition guards determines the next active steps. In a normal divergence, like (2) in Figure 2.5, can only be one branch with an active step. Simultaneously divergences allow more than one active step, (3) in Figure 2.5. The token is split at the simultaneously divergence and merged at the simultaneously convergence. Divergence and convergence are the terms of the IEC 61131. From now on, we refer to simultaneously constructs as split and merge and to the normal divergence and convergence with fork and join. There is only one global guard at a split, hence a selection of branches is not allowed.

If a token enters a step, that does not had a token before, the assigned entry actions are evaluated. The exit actions are executed, if an action is not active anymore, and active actions of a step are executed as long as the steps holds a token. If an actions is executable, because of the above rules, we call this action enabled.

SFC Definitions

There are two ambiguous behaviours in the standard. First, the action order is important, whenever multiple steps are active. Race conditions can occur, because the order of the active actions matters. Second, the branch selection of fork is ambiguous if multiple

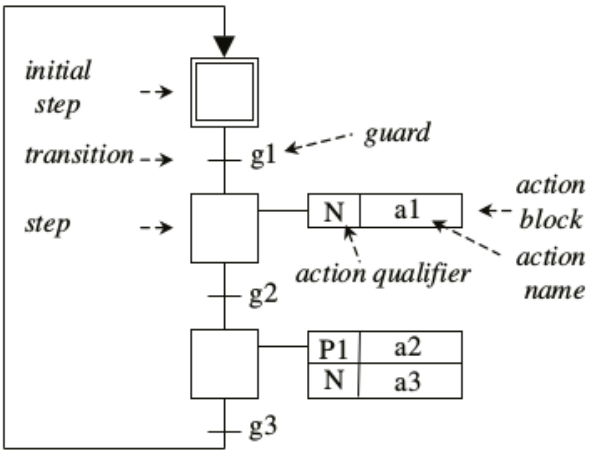


Figure 2.4 SFC with annotations from [Bau+04a]

transition guards are fulfilled. [Bau+04a] solves these ambiguity with the introduction of execution order of actions $\sqsubset$ and for the selection of transitions $\prec$. We derive our definitions from [Bor+00], with adaption to make the incorporate with other languages of the IEC 61131-3 family.

Definition 2.13 (State) *A state of a SFC is a variable assignment $\mathcal{V}$ (Definition 2.1).*

The memory $\mathcal{V}$ holds besides the declared variable of the function, the current active steps of the SFC. The state can be modified by executing the actions or taking a transition.

Definition 2.14 (Guard) *A guard g is a expression or function in any IEC language, that can be evaluated given an memory $\mathcal{V}$.*

We say a state $\mathcal{V}$ satisfies a guard g, iff g evaluates to true in $\mathcal{V}$

$$I_{\mathcal{V}}(g) = \top \text{ iff. } \mathcal{V} \models g \quad . \tag{2.54}$$

Definition 2.15 (Action, action sequences) *An action manipulates the state $\mathcal{V}$ of the SFC and is evaluable with eval.*

Additionally to [Bor+00] we qualify our actions in three categories: *entry*, *active* and *exit*. In [Bau+04a] the action is a tuple (q, n), where the qualifier q is the action category (Figure 2.4) and n denotes the name. In our case we use three functions $a_{\text{entry,active,eixt}}(\cdot)$ for the mapping between a step and corresponding action. [Bau+04a; Bor+00] explicit declare sub SFC in action sequences. We model sub SFCs as normal function blocks, that can be called like every other function block, and nested within each other in arbitrary depth. We take the definition from [Bor+00], add the categories of action and reduce their action sequence to one action w. l. g.

Definition 2.16 (Sequential Function Chart) *A sequential function chart (SFC) is a 6-tuple $\mathcal{S} =_{\text{def}} (S, s_0, T, a, \sqsubset, \prec)$, where S is a finite set of steps, $s_0 \in S$ is the initial step, $T \subseteq (2^S \backslash \{\varnothing\}) \times G \times (2^S \backslash \{\varnothing\})$ is a finite set of transitions, $a : \{active, entry, exit\} \times S \to A$ is an action labeling function which assigns an action sequence to each step, $\sqsubset \subseteq S \times S$ is*

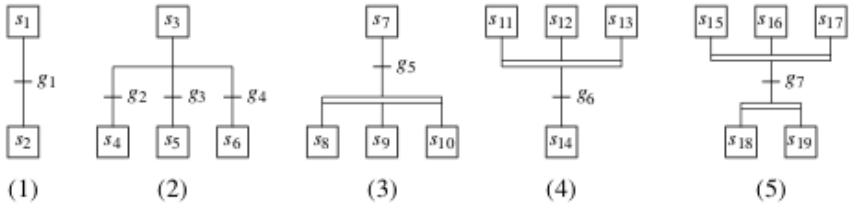


Figure 2.5 Types of possible transitions, from [Bau+04a; Bor+00]

an irreflexive total order on steps, used to define the order in which the action sequences of the active steps are to be executed, and $\prec \subseteq T \times T$ is an irreflexive partial order on transitions, such that conflicting transitions are comparable. This defines priorities for solving the conflicts.

G denotes the set of all possible guards and A the set of all possible actions. In our case, the case study, this is reduced to ST expression, respective statements.

A transition (s, g, t) connects multiple steps $s \in S$ via a guard with multiple other steps $t \in S$. The connection between multiple model split and merge of tokens. We give a transition examples in Figure 2.5. The textual description of these transitions look as follows:

1. $(\{s_1\},\ g_1,\ \{s_2\})$
2. $(\{s_3\},\ g_2,\ \{s_4\}), (\{s_3\},\ g_3,\ \{s_4\}), (\{s_3\},\ g_4,\ \{s_6\})$
3. $(\{s_7\},\ g_5,\ \{s_8, s_9, s_{10}\})$
4. $(\{s_{11}, s_{12}, s_{13}\},\ g_6,\ \{s_{14}\})$
5. $(\{s_{15}, s_{16}, s_{17}\},\ g_7,\ \{s_{18}, s_{19}\})$

Turn Algorithm

We develop the $eval_{SFC}$ algorithm for making one turn in an SFC. A turn consists of for phases:

1. Execute entry actions,
2. Execute active actions,
3. Calculate next steps and
4. Execute exit actions.

We have to execute the active entry actions (Item 1). An entry action is active if and only if a corresponding step received a token in the previous round. The next step (Item 2) is the execution of all enabled active actions. We have to calculate the successor step for each token (Item 3) by evaluating every transition outgoing from an active step. The order of evaluation is indifferent, because the guards are side-effect free. If multiple transition are enabled, outgoing from the same step, then the $\sqsubset$ order decide which transition is taken. At the end (Item 4) we execute every exit action from the steps, which will lose his token in the next round. The execution of the actions is done with respect to the action order $\sqsubset$. Going to a step, which held a token in the previous turn, does not trigger the entry action and the exit action is only trigger if the step is not active in the next turn. Hence, a self-loop over a step never triggers any entry or exit action. In each turn, every token is able to take one transition, but only if the associated guard is satisfied. Tokens can only be created on splits, respective destroyed on merges. Figure 2.6 execute one turn of a SFC.

Data: A SFC $\mathcal{S} =_{\text{def}} (S, s_0, T, a, \sqsubset, \prec)$.
Input: A variable assignment $\mathcal{V}$, a set of active steps $\mathbb{S} \subseteq S$ and a function $new\colon S \to \mathbb{B}$, that determines if the step was marked as active in the last turn.
Output: A variable assignment $\mathcal{V}'$, a set of active steps $\mathbb{S}' \subseteq S$ and a function $new'\colon S \to \mathbb{B}$, that determines if the step was marked as active in this turn.
begin
  $\mathcal{V}' := \mathcal{V}$;
  /* Execute entry actions of fresh marked steps */
  for $step \in \mathbb{S} \wedge new(step)$ *in order of* $\sqsubset$ **do**
    $\mathcal{V}' := eval(\mathcal{V}', a(\text{entry}, step)$;
  end
  /* Execute active actions of active steps */
  for $step \in \mathbb{S} \wedge step \in \mathbb{S}$ *in order of* $\sqsubset$ **do**
    $\mathcal{V}' := eval(\mathcal{V}', a(\text{active}, step)$;
  end
  /* Calculating transition active transitions */
  $T' := \{(x, g, y) \mid x \subseteq \mathbb{S} \wedge I_{\mathcal{V}'} \models g\}$;
  /* Decide conflicts by transition order $\prec$ */
  for $(t_1 = (x, g, v)) \in T' \wedge (t_2 = (x, g', v')) \in T'$ **do**
    if $t_1 \prec t_2$ **then**
      $T' := T' \setminus \{t_1\}$
    else
      $T' := T' \setminus \{t_2\}$
    end
  end
  $\mathbb{S}' = \mathbb{S}$;
  $new' = \varnothing \times \varnothing$;
  for $(x, g, y) \in T'$ **do**
    $\mathbb{S}' := (\mathbb{S}' \setminus x) \cup y$ /* forward token from x to y */;
    /* marking new active steps */
    $new'(step) := \top$ for all $step \in y \wedge step \notin \mathbb{S}$;
  end
  /* Execute exit actions of leaved steps */
  for $step \in \mathbb{S} \setminus \mathbb{S}'$ *in order of* $\sqsubset$ **do**
    $\mathcal{V}' := eval(\mathcal{V}', a(\text{exit}, step)$;
  end
end

Figure 2.6 $Turn_{\mathcal{S},\mathcal{V}}(\mathbb{S}, new)$: This algorithm evaluates one turn for a SFC $\mathcal{S}$

Data: A SFC $\mathcal{S} =_{\text{def}} (S, s_0, T, a, \sqsubset, \prec)$.
Input: A variable assignment $\mathcal{V}$, a set of active steps $\mathbb{S} \subseteq S$ and a function $new\colon S \to \mathbb{B}$, that determines if the step was marked as active in the last turn.
Output: A variable assignment $\mathcal{V}$, a set of active steps $\mathbb{S}' \subseteq S$ and a function $new'\colon S \to \mathbb{B}$, that determines if the step was marked as active in the last turn.

```
if V(SFCReset) then
    S := {s_0}                    /* Jump into initial step */;
    Reset all SFC control flags in V;
    Set new(step) := ⊥ for all step ∈ S;
else
    if V(SFCInit) then
        S := {Init}               /* Jump into initial step */;
        Set new(step) := ⊥ for all step ∈ S;
    end
    if ¬V(SFCPause) then
        return Turn_S(V, S, new)
    else
        return V, S, new
    end
end
```

Figure 2.7 $Turn'_{\mathcal{S}}(\mathcal{V}, \mathbb{S}, new)$ Turn algorithm with controllabe flow

Control Flags

Several special variables controls the SFC's control flow. These variables are vendor specific and the code of the case study [VH+14] uses these variables offered by CODESYS, for handling of the emergency stop. We restate the behaviour of SFCInit, SFCReset and SFCPause from the internal CODESYS help.

SFCInit If this variable gets true, the sequential function chart will be set back to the initial step. All steps and actions and other SFC flags will be reset (initialization). The initial step will remain active, but not be executed as long as the variable is true. SFCInit must be set back to false in order to get back to normal processing.

SFCReset This variable behaves similarly to SFCInit. Unlike the latter however, further processing takes place after the initialization of the initial step. Thus in this case for example a reset to false of the SFCReset flag could be done in the Init step.

SFCPause As long as this variable is true, the execution of the SFC is stopped.

Figure 2.7 wraps the standard $Turn_{\mathcal{S},\mathcal{V}}(\mathbb{S}, new)$ function for providing support of the control flags. The interface stays the same and the control flags are given over the variable assignment $\mathcal{V}$. Setting SFCReset and SFCInit does not lead to an execution of the entry and exit actions of the active steps.

We can now state the $eval_{SFC}$ function.

Definition 2.17 ($eval_{SFC}$)

$$eval_{SFC}(\mathcal{V}, \mathcal{S}) =_{\text{def}} \quad \mathcal{V}' \tag{2.55}$$

$$\text{in} \quad \mathcal{V}',\ S_{t+1},\ new_{t+1} = Turn'_{\mathcal{S},\mathcal{V}}(\mathbb{S}_t, new_t), \tag{2.56}$$

where $\mathbb{S}_t$ and new_t are the variables from the previous call with $\mathbb{S}_0 := \{s_0\}$ and $new_0(x) := \bot$ for all $x \in S$.

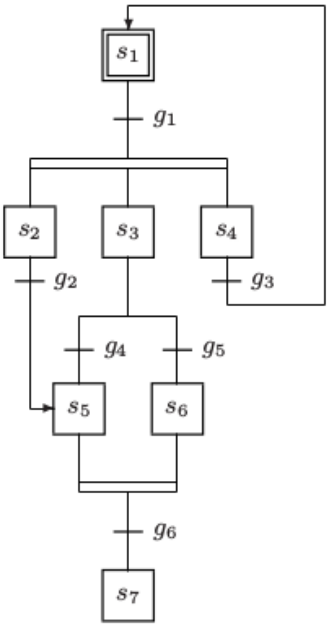


Figure 2.8 A not well-formed SFC from [Bau+04a]

Well-formed SFC

Not every SFC defined by Definition 2.16 is well-formed in the sense, that is behaviour is clear during the execution. Figure 2.8 gives an example. [Bau+04a] restrict themselves to just "safe" SFC, but leaving out a proper definition. In Figure 2.8 there are three cases, which cause an unclear behaviour. First, the transition with guard g_3 creates unlimited tokens. Second, tokens can accumulate in step s_5 and a step can only hold one token at a time. Third, the split creates three tokens, but only two tokens can reach the merge, hence step s_7 is not reachable. We want to narrow the definition of not well-formed SFCs, knowing that there is no absolute definition of this. We state a SFC is well-formed, if it is clear and determined how to execute the state chart.

Every SFC is well-defined if it does not contain split or merge. Without split and merge, there is only one token, resp. one active step, at every execution step. With the prohibition of split and merge we are near the deterministic finite automata.

Lemma 2.18 (Well-formed SFC without simultaneous constructs) *A SFC* $\mathcal{S} = (S, s_0, T, a, \sqsubset, \prec)$ *is well-formed, if does not contain split and merge, hence*

$$\forall\,(x, g, y) \in T\colon |x| = 1 \wedge |y| = 1$$

But not every SFC, that uses split and merge is miss-formed. We describe the set of well-defined SFC further. We address the safe use of split and merge and the prevention of unlimited tokens, like in Figure 2.8. SFC with split and merge is safe, if it is guaranteed at every time, that token count is limited This is not sufficient as Figure 2.9 presents an counter example. If we set b_2 to $\bot$, we could overcome the merge and leave a token behind in step 22. Additionally step 23 could hold two tokens.

Unlimited tokens can only be created with transition, that breaks out the split-merge section, like in Figure 2.8. Interference between branches in split-merge section is able to create multiple tokens in a step. We create partitions on the steps of a SFC and claim the isolation of the found partition. There can at most one token in each partition. Only split and merge transitions are allowed between token partitions. Any normal transition violates the prohibition of interference between the branches of a split-merge section.

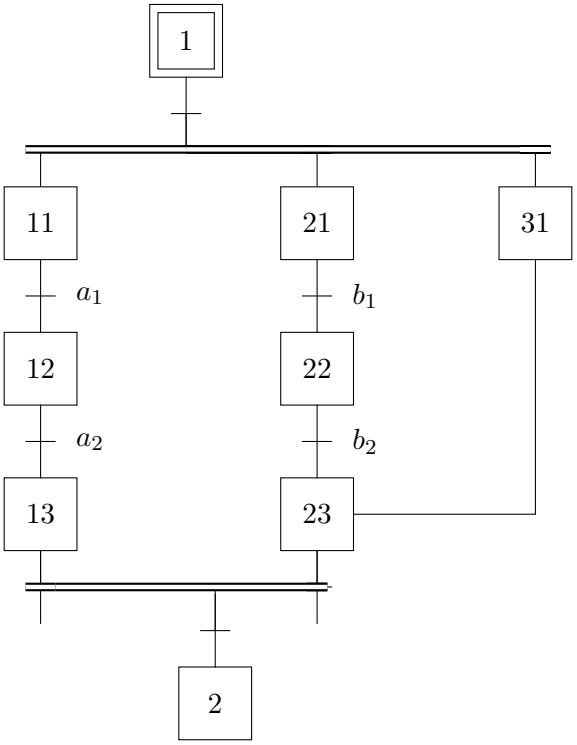


Figure 2.9 Unsafe SFC with restriction of token amount

Definition 2.19 (Token Partition) *A token partition is a set $P_T \subseteq S$ for a SFC $\mathcal{S} = (S, s_0, T, a, \sqsubset, \prec)$. The token partition fulfills following properties:*

- *In a token partition there exists at most one token (active step).*
- *There are only normal transition between steps in a token transition.*

$$\forall a, b \in P_T : \forall (s, g, t) \in T : a \in s \wedge b \in t \rightarrow |s| = 1 \wedge |t| = 1$$

A partition starts with the initial step or the first step after a split and is limited by next merge. Every step, that is reachable via normal transitions, fork and join belongs to one token partition. Figure 2.10 show a schematic SFC, where we mark the token partitions with colors. But the SFC in Figure 2.10 is not safe, althoug the token partitions are disjoint.

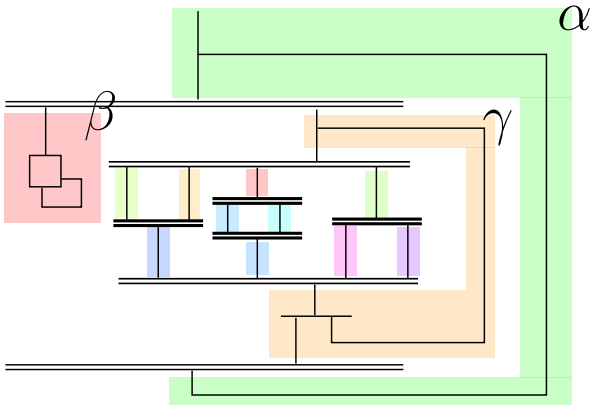


Figure 2.10 Token Partitions of a schematic SFC. α is the outer partition of β and γ.

It is still possible to create an arbitrary token quantity, the tokens in the left branch from top split never reach the merge and we can visit the top split unlimited time.

Lemma 2.20 (Well-formed simultaneous SFC) *Let $\mathcal{S} = (S, s_0, T, a, \sqsubset, \prec)$ be SFC with simultaneous constructs and $p_1, \ldots, p_n$ token partitions, then the SFC is well-defined, if*

$$p_i \cap p_j = \varnothing$$

for $0 \leqslant i < j \leqslant n$ and if a step in a outer partition is active, no step in an inner partition is active.

The relation of outer and inner partition is defined over the transitions. Let P_o a outer partition to the inner partition P_i, then

outer-inner partition

$$\exists (s, g, t),\ (s', g', t') \in T \colon |s| = 1 \wedge |t| > 1 \wedge |t'| = 1 \wedge |s| > 1 \tag{2.57}$$
$$\wedge\ \exists a \in s \colon a \in P_o \tag{2.58}$$
$$\wedge\ \exists b \in t \colon b \in P_i. \tag{2.59}$$

Equation (2.57) describes, that we can reach the inner partition via split from the outer partition and merge leads from the inner to the outer partition. The algorithm for finding the token partition is based upon breadth first search, maintaining an history of created tokens by splits. We give a sketch in Figure 2.11. The algorithm maintains *reached*, $prefix$ and *weight*. *reached* contains the current reached steps and enables transitions. $prefix$ contains the partition as a list of numbers, the numbers maintain the hierarchy and history of created partitions. *weight* determines, if a split-merge section is closed. We claim, that $prefix(a)$ returns of $a \in S$ or the common prefix if $a \subseteq S$.

The algorithm stops if every step is reached. The second loop considers all transition, which outgoing steps has been reached. For a normal transition we copy the prefix and weight. On splits, we share the weight among the branches and create a new token identifiers with prefixes from the outgoing step. On merge, we sum up the weights to determine, if the section is closed. On a closed section we use the common prefix between all outgoing branches, else we leave one branch open. The algorithm is correct, if all created tokens will finally reached a merge and the given SFC $\mathcal{S}$ is safe. If the SFC is not safe, the algorithm assigns two different prefixes to one step.

If the token partitions of a SFC do not overlap, the SFC is safe. Every step is in a token partition:

$$p_1 \cup p_2 \cup \cdots p_n = S$$

We can not state the other direction in Lemma 2.20, because the definition of safe SFC is not formally described. It is more a notion created by examples of an undefined behaviour. The definition of token partitions helps us by the reduction from SFC with split and merge into deterministic SFCs (Proposition 2.22).

2.4 $\mathbf{ST_0}$

We build an intermediate representation (IR) for the regression verification. The IR ST_0 is a subset of ST. We need a transformation of SFC into ST and transformation for making

```
Input: SFC S = (S, s_0, T, a, ⊏, ≺)
Output: Partition identifier prefix.
begin
    reached := {s_0};
    prefix(s_0) := ⟨0⟩;
    weight(s_0) := 0;
    while reached = S do
        for t = (u, g, v) ∈ T if all u ⊆ reached do
            T = T\{t};
            reached := reached ∪ v;
            if |u| = |v| = 1 then                          /* Normal transition */
                weight(v) := weight(u);
                prefix(v) := prefix(u);
            else
                if |u| = 1 ∧ |v| > 1 then                              /* split */
                    i := 0;
                    for a ∈ v do
                        prefix(a) := prefix(u) ∘ i;
                        weight(a) := weight(u) / |v|;
                        i := i + 1;
                    end
                else                                                   /* merge */
                    weight(v) := Σ_{a∈v} weight(a);
                    if weight(v) ≠ weight(a) for a step a with prefix(a) = prefix(u)
                    then /* old weight not reached, hence section not closed,
                    open new token partition */
                        prefix(v) := prefix(u) ∘ 0;
                    else                              /* use the common prefix */
                        prefix(v) := prefix(u);
                    end
                end
            end
        end
    end
end
```

Figure 2.11 This algorithm finds token partition in an SFC.

ST simpler. Figure 2.12 shows the translating steps for ST and SFC language into ST_0. The other IEC 61131-3 languages can also translatable into ST and could be embedded in this workflow, but are out of scope for this thesis.

ST_0 is a restriction on ST. The operational semantics (Definition 2.11) stays the same. We only prohibit certain statement and expressions in Definition 2.21

Definition 2.21 (ST_0) *ST_0 follows Definitions 2.5, 2.7 and 2.9 to 2.11 The generation ST_0 program starts with the $\langle ST \rangle_0$ non-terminal.*

$$\langle ST_0 \rangle \rightarrow \langle type \rangle \tag{2.60}$$
$$\text{'PROGRAM'} \langle name \rangle \tag{2.61}$$
$$\langle los_0 \rangle \tag{2.62}$$
$$\text{'END\_PROGRAM'} \tag{2.63}$$
$$\langle los_0 \rangle \rightarrow \langle statement_0 \rangle \text{ (';' } \langle statment_0 \rangle)^* \tag{2.64}$$
$$\langle statement_0 \rangle \rightarrow \langle assign \rangle \mid \langle if \rangle \tag{2.65}$$

In ST_0 the following data types can be used

- *integer (SINT, INT, DINT, LINT),*
- *unsigned (USINT,UINT, UDINT, ULINT),*
- *enumerations and*
- *boolean.*

Function calls within expressions are prohibited.

Only one program definition and one type declaration is allowed in a ST_0 program. The function and function blocks calls have to be embedded in the program body. For loops need to be unwound. While and repeat loops are not supported in the translation. Loops are avoided in ST, because they can diverge and the program does not terminate at the end of scan cycle. So real software spare the use of while and repeat loops, like the software in the case study. Unfolding of structs and arrays helps on restriction of data types. The only prohibited data structures are the strings and reals, as dates and times can be compose as integers. Later in Section 3.3.1, we present the transformation of ST_0 to a SMV module.

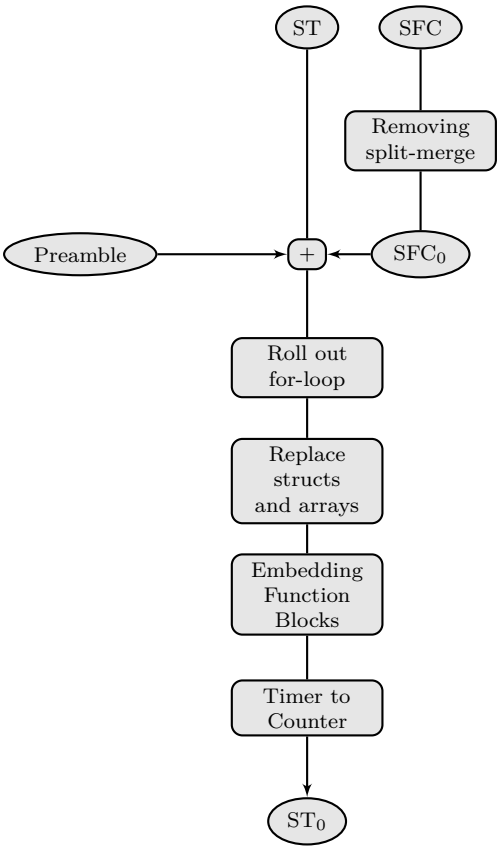


Figure 2.12 Preprocessing for ST_0

2.4.1 $\mathbf{SFC_0}$

We want to purify SFC from unambiguous and complex behaviour. Our target is to translate well-defined SFC into a deterministic form (Lemma 2.18) and with disjoint guards on forks. In the end we do not need the order of actions $\sqsubset$ and translation $\prec$ anymore.

We ensure the disjunction of the guards by simultaneously replacing the guards ([Bor+00]):

$$g' =_{\text{def}} g \bigwedge_{t_1=(Q_1,g_1,Q_1')\in T, Q_1\cap Q_2\neq\emptyset, t_1\prec t} \neg enab(g_1) \quad . \tag{2.66}$$

There is no need for the transition order $\prec$ after rewriting the guards.

The removal of merge and split sections is more complex. We put the different token partitions (Definition 2.19) within split and merge into separate SFC function blocks. A new step replace the split and merge section and calls the created SFC function blocks. The attention lies upon entry and exit in the token partitions. We have to keep the same behaviour, for every turn. If such token partitions exists, we can rely on the disjointedness of the split branches.

Every partition, becomes a separate function block and the upper split-merge section is replace through a proxy step in the top SFC. The top SFC is the outer partition, and the inner partition becomes a sub SFC. In the Figure 2.11 we can decide by investigating the prefix of both partitions identifiers. The new function block need to access every variable from top SFC. Additionally we need to reset the sub SFC (partition) if we enter proxy step. There is one point we loose in the transformation. This abstraction preserves the action order $\sqsubset$ only if the order is consistency in one partition. Hence, every associated action a_i from partition p_i is executed before the associated actions from partition p_j. If we reduce the partitions to a proxy step, we can only decide the execution on partition level. There may be complex ways to preserves the action order. This is sufficient as the vendor specifies the order of the action execution branch-wise from right to left, resp. Siemens from right to left.[4]

Proposition 2.22 (Elimination of simultaneous constructs) *Let $p_1, \ldots, p_n$ be token partitions of split-merge section with the incoming transition $t = (x, g, y) \in T$ and the outgoing transition $t' = (x', g', y') \in T$ in a given SFC $\mathcal{S}$.*

1. *Every p_i becomes a SFC τ_i with the first reached step after a split as the initial step.*
2. *Each variable from $\mathcal{S}$ is duplicated into τ_i as a* 'VAR_IN_OUT'*.*
3. *We add a new variable accept in every τ_i as a signal, that the SFC has reached the last steps before the convergence. The last steps l_j in each SFC τ_i sets accept to $\top$.*
4. *We replace the whole split-merge block by a new step ρ. $a(entry, \rho)$ resets every SFC τ_i and calls the entry action from the seed step.[5] The active actions calls every function block τ_i. Before and after each call the scope of $\mathcal{S}$ and τ_i are synced by setting the variables in τ_i (resp. $\mathcal{S}$). The function calls block are in order of $\sqsubset$. Every exit action of the last steps from the sub SFC τ_i is gathered into the λ's exit action in order of $\sqsubset$.*
5. *The outgoing transition $t' = (x', g', y') \in T$ from the split is replaced by new transition $(\{\rho\},\ g' \wedge \bigwedge_{1\leqslant i\leqslant n} accept_i,\ y')$.*

[4] Covering complex orders: Imagine, the functions blocks of the sub SFC return a set of action identifiers and the proxy step executes each action in the returned set in the in the specific order.

[5] The entry action of initial steps not called in the start.

We need clarify some parts of Proposition 2.22. First the proposition should applied to the most inner pair of split and merge. An application of the proposition does not lead to miss-formed SFC and the behaviour is not changed. We apply the proposition repeatedly until every split and merge has disappeared. We just have to be careful in the selection of the token partitions. If we want to proof the soundness of the construction, we need to limit the $\sqsubset$, as discussed above. We define an order $\preccurlyeq$ on the token partitions with

$$Q \preccurlyeq P \quad \text{iff} \qquad \forall q \in S,\ p \in P\colon \quad a_Q(q) \sqsubset a_P(p)\ , \tag{2.67}$$

where Q, P are token partition. Only if every action from Q executed before P in the original SFC we can safely put proxy steps in charge.

For the proof, we fixate a split-merge block and compare at the entry, active and exit in the token partition and the proxy step. We need to assume that Equation (2.67) holds pairwise for every token partition in a split and merge section. The goal is to show, that the entry, active and exit section for each step is activated in the same step if the same input comes in, which results in the execution same actions.

Proof 2.23 (Soundness of Proposition 2.22) *We fixate a split-merge section with an incoming transition t and a outgoing transition s.*

entry turn *The transition $t = (x, g, \{y_1, \ldots, y_n\})$ into the split-merge block is satisfied $\mathcal{C} \models g$ and the exit action $a(exit, x)$ is called for the step. In the next step following happens:*

- ***Old behaviour*** *We jump directly into every step $y_1, \ldots, y_n$ after the split. First the entry action $a(entry, y_i)$ is called, then the active action $a(active, y_i)$ – both categories in the order of $\sqsubset$. After finding any successor, the particular exit actions are called.*
- ***New behaviour*** *The new step ρ get the token, $a(entry, \rho)$ is called, and resets the sub SFCs and calls the $a(entry, \tau_i)$. In $a(active, \rho)$ the Figure 2.6 is invoked on all sub SFCs. It calls on every sub SFC $a(active, y_i)$ and maybe $a(exit, y_i)$. The exit action is only called if the transition is satisfied after the active action. The same behaviour as in the old SFC. The actions are called in $\preccurlyeq$ order after assumption.*

active turns *The new proxy step just forwards every turn in the top SFC to the sub SFC. The order of steps in both behaviour are the same and the proxy can only be left if every old guard on transition s holds and every sub SFC has* accepted*.*

exit turn *Let assume the tokens are merged via the transition $s = (\{x_1, \ldots, x_n\}, g, y)$. This happens iff all started token at split reached a step before the merge, and g is valid. This is the same condition as in the guard after the proxy step. We assume the leaving of the split-merge section.*

- ***Old behaviour*** *If we leave the section, we are in the last steps x_i of each token partition and notice, that we take the transition. $a(active, x_i)$ is called for the last time and $a(exit, x_i)$ once respecting the order $\sqsubset$. In the next step $a(entry, y)$ and $a(active, y)$ are called.*
- ***New behaviour*** *Proxy step ρ is active. $Turn(\cdot)$ executes in $\preccurlyeq$ order the sub SFC. For every sub SFC τ_i, the last step x_i is active and $a(active, x_i)$ is executed. So $accept_i$ becomes valid.*

The upper SFC evaluates the transition guard. in the upper SFC is checked The guard is valid because g (assumption) and $\bigwedge_{1 \leqslant i \leqslant n} accept_i$ *is valid. We are leaving* ρ *and call* $a(exit, \rho)$*. After Proposition 2.22* $a(exit, \rho)$ *contains every exit action from the sub SFC*[6]*.*

In the next step $a(entry, y)$ *and* $a(active, y)$ *are called.*■

Of course is the order of action execution differently, but the changed variables appeal at the end of a scan cycle. The execution order matters only internally if variable are dependent. For the transformation to ST_0, we assume that the action sequences (entry, active, exit) are encoded into words over $\mathcal{L}(\langle los \rangle)$, so there are valid ST code.

Definition 2.24 (SFC_0) *A SFC_0 is a SFC with following restriction:*

- *every action sequence is coded in ST*
- *no splits and merges*
- *disjoint guards on forks*

2.4.2 Transformation of SFC_0 into ST

For the transformation from SFC_0 to ST we only need to roll out the Figure 2.6 for a given SFC $\mathcal{S}$. We give a template to generate the ST code in Figure 2.13.

Template Language

The template language is like a macro, that generates code on execution. The placeholder are in angle brackets and part of the meta level. We replace a placeholder `<a>` by the value of the a with supports of fields. The for loop `<for a in seq>` duplicates the content between `<for>` and `<endfor>`, and assigns a value from *seq* to a. See Section 2.5 for examples.

Template SFC to ST

We first create a new enumeration type `<name>_state_t`, where `<name>` is the name of the SFC (or some arbitrary unique identifier). Differently, we could encode the steps as integers, but for debugging and tracing we use the step names. The SFC becomes a function block with the same name `<name>`. Additionally, the old variables `<old_variables>` from the SFC we declare two more: `_state` holds the current step, and `_transit` is valid iff a transition is taken. For every step `<s>` we create an entry in a case statement. The entry action `<s.entry>` is called if the step is freshly encountered, hence `_transit` is valid. After entry action we reset `_transit` and call the active action `<s.active>`, which is always executed. Every outgoing transition is represented by an **`ELSEIF`** branch. The order doesn't matter after the replacement of the guards in SFC_0. If a transition guard `g` is satisfied, we need to set the new state `v` and activate `_transit`, so the entry function `<u.entry>` will be called in the next round. Besides, `_transit` triggers the exit action `s.`**`exit`**. The Figure 2.13 covers the Figure 2.6.

2.4.3 Transformation of ST into ST_0

To bring ST code into ST_0 form, we apply following transformations:

- unwind loops
- unpack structs and arrays
- timer to counter
- embedding function blocks.

Every transformation can applied on it's own. But there are good and bad orders. We separately present the transformation. For the same transformation we use the same template language as in Section 2.4.2.

[6] The exit action are in the top SFC, because we can not decide to execute them locally in the sub SFC.

Unwinding of for loops

A for loop has the form of Equation (2.34). We need statically evaluable loop boundaries <start>, <stop> and <step>, that means the evaluation of the boundary is possible on the basis of the syntax. We support only expression that composed of literals and constants variables. Additionally, we have to ensure that the loop variable <var> is not written within the loop body. Otherwise the termination of the for loop is not guaranteed and the unwind is not sound.

In the transformation, the for loop is pulled up to the macro level (Figure 2.14). With `start:step:stop` we refer to a sequence

$$a_n =_{\text{def}} n \cdot_M step +_M start$$

for every loop iteration i until $a_i > stop$ and $*_M, +_M$ denotes the multiplication and addition in machine semantics. The evaluation of the sequence has to be done within the boundaries of IEC 61131-3's data types. In the loop body <body> we replace every occurrence of the loop variable with current constant value. The replacement helps us in the next steps if we unfold structs and arrays. The counter variable is defined in procedure's scope, so we need to assign the first value beyond the boundary to counter variable.

Replace structs and arrays

Structs and arrays are just composition of data types. In structs the sub elements are addressed by name and in arrays by multiple indices. In both cases the amount of sub elements are determined statically and can not be changed during runtime.

Structs

We rewrite every access to an element <var>.<element> of a struct variable into <var>_<element>, and declare the variable <var>_<element> in the scope of the old struct variable. Of course the used variable should not already exists, else we can generate a other unique name. We can apply the same technique repeatedly on struct variables, to flatten the hierarchy.

Arrays

The access to array elements happens over integer expressions, for example `a[ floor(sin(x)+1)]`. Like the for loop boundaries the expressions have to be evaluable syntactically. For this reason, we have replaced the loop counter variable with a constant in the loop body. The replacement is the same as with structs, if we find an array access, for example `name[a,b]`, we calculate the accessed positions a, b and rewrite it as normal a variable <name>_<a>_<b>. In the variable scope we need to introduce every possible position of the arrays.

We need to consider the case of an assignment of whole structs or arrays variables, or calling functions with the variables. The assignment of composite data types, are assignments of every element recursively. We can easily expand such an assignment to an assignment of every element. The input and output parameters in function block applications becomes assignments before and after the call. This does not work for functions. Anyway we prohibited function calls in expression, by the definition of ST_0.

We can use the unfolding of data structures for covering the built-in data structures of IEC 61131-3, too. For example, we can define the date data types like Figure 2.15 in the preamble. Additionally, we need to reimplement the built-in functions, before these data types become useful.

Timer to counter

IEC 61131-3 defines timer function blocks, that depends on the real time of the PLC. Real time is hard to model into model checkers (see [Lam02] for examples). We are simulating the system in scan cycle turns and assume that every turn consumes a certain time quantity. So instead of counting the time, we count the scan cycles.

We look at the TON function block, because the case study uses it. TON has two input argument IN, PT of BOOL and TIME, and two output variables Q, ET of bool and TIME. Informal, TON set Q to true if the given time PT has run out and IN stays valid at all calls. In other words, Q becomes true iff IN is valid at least PT seconds long. ET hold the time since the timer has triggered. TON is often used to wait a specific time before continuation of the control flow. Formal we can describe TON as a function. Note the normal behaviour specification is informal given as timing diagrams [TJ09].

$$TON(IN, PT) = \begin{cases} \bot, \bot & : IN = \bot \\ \bot, 0 & : IN' = \bot \wedge IN = \top \\ \bot, ET + CCT & : IN' = \top \wedge IN = \top \wedge PT < ET \\ \top, PT & : IN' = \top \wedge IN = \top \wedge PT = ET \end{cases}, \tag{2.68}$$

where CCT is the time quantity consumed by one scan cycle, and IN' is the IN input from the previous call. TON returns a tuple of Q, ET, where Q becomes true after the waiting time ET reaches PT. ET is limited to PT. There is no definition for the behaviour if changes PT during several calls with IN is true. The difference to a counter based approach, is setting CCT to one and PT is divided through the CCT. A function block definition for TON could be like Figure 2.16. Later in Section 3.3.3 we will discuss a technique for getting complete rid of TON.

Embedding function blocks

The embedding of function blocks consists of multiple steps:

1. create a copy of the function block with prefixed variables.
2. duplicate the function blocks variables in the caller scope
3. replace every call to this function block with the block's statements.

The prefix is the instance name followed by unique terminator, for example "$". We have to check for variable name collisions and need to set the input and output before and after the call. The application may have to be done multiple times until every function block call disappears. Mutual use of function blocks is not allowed after IEC 61131-3.

2.5 Software for Introductory Example

We pickup the introductory example (Section 1.1). First, we create the SFCs for both revisions and a main program. Second, we show the transformation into ST_0.

SFC Revision I

The SFC for revision I is in Figure 2.17a . We start in step *Wait* with a non-moving belt and wait there, until a workpiece is lay on the belt. The sensor w_1 registers the appeared workpiece ($w_1 = \top$) and we jump into *Run* step. We wait in *Run*, that the activated the conveyor belt ($run := top$) has transported the workpiece to the right edge ($w_2 = \top$). Then SFC goes into *Pick*, stops the belt ($run := \bot$) and activates the crane (*pickup*). After the crane removed the workpiece ($w_2 = \bot$), we wait for next workpiece. Interestingly, there is a

synchronize mechanism, like barriers or mutual exclusion, via the real physical world. One process is halted until a physical event occurs, that is created by another process within the software or the physical world. In our example the belt halts, when the workpiece hits w_2.

SFC Revision IIa, IIb

We want to extend our first revision from Figure 1.2 by a sorting feature. Figure 1.3 shows the new constellation with the new detector D. The detector can distinguish correctly manufactured workpieces from broken ones during the transportation. The crane should only pick up correct workpieces, while the broken workpieces fall over the right edge into the wasted bin. Figures 2.17b and 2.17c shows two possible solutions for the second revision. In revision IIa (Figure 2.17b) the new behaviour introduces a branch. The new branch prevents the belt from being halted, if the detector saw a broken workpiece. Revision IIa is not minimal. In revision IIb, the same behaviour was achieved with an more complex action with in the step *Pick*. Obviously, revision IIa and IIb behave the same under equal input w_1, w_2.

main program

We need a program to call our SFC function blocks. The program in Figure A.1 takes care about emergency stop, forwards sensor values to SFC and sets the actuator commands from the SFC into the bus. The mapping between the local program variables and the bus system is done on the configuration level. Both the function block and the program does not contain any complex data types or loops. We can directly embed the function block into the program (Figure A.3). Note, the case statement becomes an if statement and every variable from the function block is copied with a prefix into the program. We have leave out the empty entry actions and the transit variable.

```
1  TYPE <name>_state_t : (<state_names>); END_TYPE
2  FUNCTION_BLOCK <name>
3    VAR _state   : <name>_states_t ;
4        _transit : BOOL
5    END_VAR
6    <old_variables>
7
8    CASE _state OF
9      < for s in S >
10     <s.name>  :
11         IF _transit THEN <s.entry> END_IF; {* entry action on transit*}
12         _transit := FALSE;
13         <s.active> {* active action always *}
14
15         IF FALSE THEN {* empty then branch *}
16         < for (u, g, v) in T where u = s >
17         ELSEIF <g> THEN
18            _state := <t>;
19            _transit := TRUE;
20         < endfor >
21         END_IF;
22         IF _transit THEN <s.exit> END_IF; {* exit action on transit*}
23     < endfor >
24   END_CASE
25 END_FUNCTION_BLOCK
```

Figure 2.13 Generation of the Figure 2.6 for a specific Sequential Function Chart 0 $\mathcal{S} = (S, s_0, T, a, \sqsubset, <)$ with a macro language.

```
1  FOR <var> := <start>
2      TO <stop> [ BY <step> ]
3  DO <body>
4  END_FOR
```

```
1  < for i in start:step:stop >
2    <body[var/i]>
3  < endfor >
4  <var> := (i+1)*step+start;
```

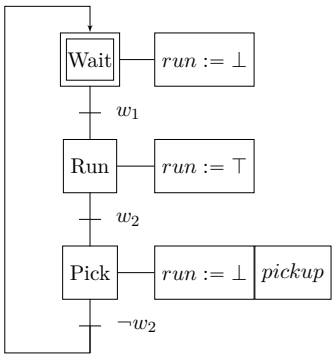
Figure 2.14 Unwinding pattern for for loops. The for loop is pulled to the macro level with evaluated boundaries. The body is copied with the replacement of the loop variable by a constant.

```
1  TYPE
2      DATE : STRUCT
3        day   : USINT (1..31);
4        month : USINT (1..12);
5        year  : UINT;
6      END_STRUCT;
7
8      TIME_OF_DAY : STRUCT
9        hour    : USINT (0..23);
10       minute  : USINT (0..59);
11       seconds : USINT (0..59);
12       ms      : INT   (0..999);
13     END_STRUCT;
14
15     DATE_AND_TIME : STRUCT
16        d : DATE;
17        t : TIME_OF_DAY;
18     END_STRUCT;
19
20     TIME : STRUCT
21       days         : INT;
22       hours        : INT;
23       minutes      : INT;
24       seconds      : INT;
25       milliseconds : INT;
26     END_STRUCT;
27 END_TYPE
```

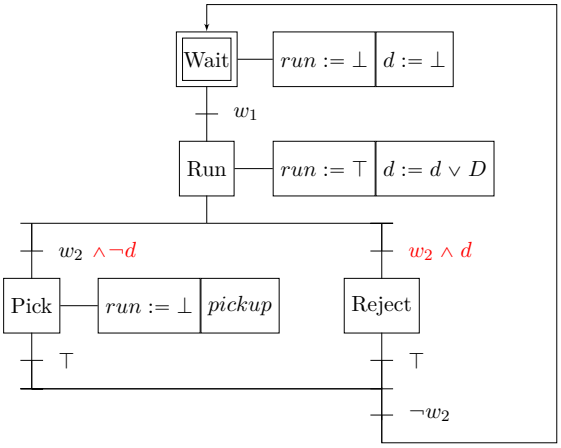
Figure 2.15 Preamble for some built-in IEC 61131-3 data types. Time is often encoded as one integer value.

```
1  FUNCTION_BLOCK TON
2    VAR_INPUT  IN : BOOL; PT : USINT; END_VAR
3    VAR_OUTPUT Q  : BOOL; ET : USINT; END_VAR
4
5  IF IN THEN
6      Q := ET = USINT#0;
7      IF ET > USINT#0 THEN
8          ET := ET - USINT#1
9      ELSE
10         ET := USINT#0
11     END_IF
12 ELSE
13     Q  := FALSE;
14     ET := PT;
15 END_IF;
16 END_FUNCTION_BLOCK
```

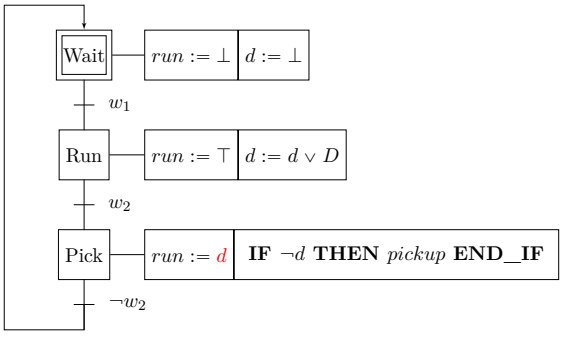
Figure 2.16 Function block for simulating TON component



(a) SFC for Revision I



(b) SFC for Revision IIa



(c) SFC for Revision IIb

Figure 2.17 Revision of the SFC for our example

3. Regression Verification

Regression verification is the assurance of program equivalence. In this section we close the gap between our intermediate representation language ST_0 and verification of equivalence (Section 3.3). We present theoretical and soundness consideration about the equivalence between of plants controlled by PLC (Section 3.2), and abstract the whole plant in general as a cyber-physical system (Section 3.1). Additionally to the transformation of ST_0 to Symbolic Model Verifier (SMV), we show the encoding of the equivalence notion into invariants and define templates for modelling the *claims*, operator on SMV level, for modelling relaxing equivalence in different ways.

This chapter provides, besides the transformation and theoretical aspects, two impulses for making verification friendly for engineers. First, we define equivalence *claims* as high level operator for equivalence statements in Section 3.4. Second, in Section 3.5 we deduce conditions under which the equivalence holds. This works on symbolic bisimulation of the same SFC in both revisions and performs well on the case study.

3.1 PLC as cyber-physical systems

Cyber-physical systems (CPS) are about the collaboration of physical entities controlled by software. The physical entities are part of and controlled by electronic and mechanical components. In our case the CPS contains the PLC, actuators and sensors (Figure 3.1),

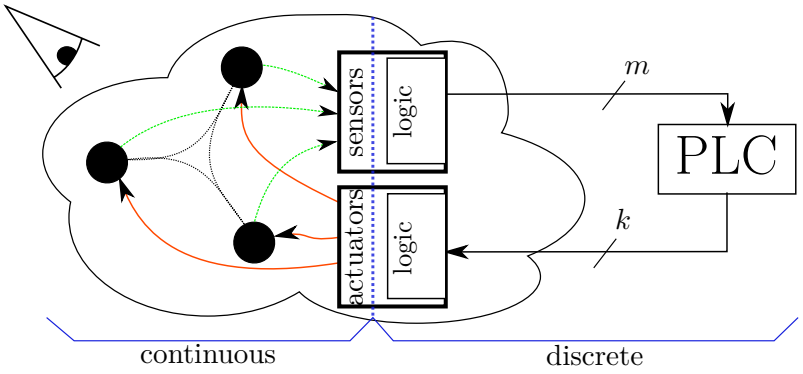


Figure 3.1 Illustration of the components in a cyber-physical system controlled by a PLC. The PLC communicates via the bus system with the logic circuits in the sensors and actuators. The sensors transforms the physical magnitudes into discrete values, vice versa for the actuators.

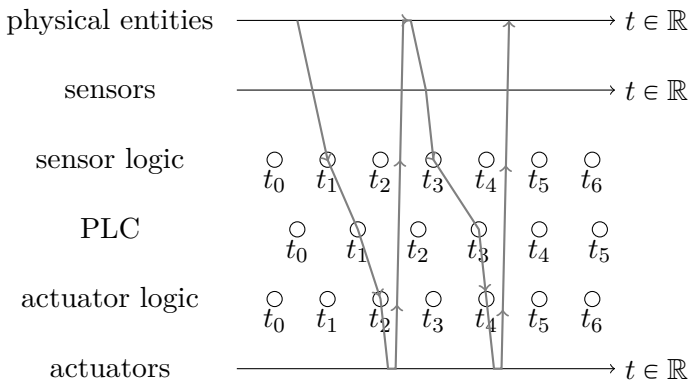


Figure 3.2 Timing and communication behaviour of CPS PLC model

discrete vs. continuous

and a central control logic software runs on the PLC. We divide the model into an discrete and a continuous world. In the discrete part the state is digital, including elements from enumerable and finite sets. The state consists the memory of the actuators and sensor logic, the PLC memory and the actual values on the bus system. Therefore the continuous world covers the physical entities, like velocity, acceleration, temperature or radiation, and often contains states from uncountable sets like $\mathbb{R}$. The state is built up implicit from actual position of workpieces, motors, current or voltages. The physical entities are modelled with differential equation. In despite, a model for the discrete world comprises logical and arithmetical expressions. Sensors and actuators are in both world. One part communicates with the PLC with discrete values. The other part read or manipulate the physical magnitudes.

PLC

The PLC is the central controller of the hardware components by sending commands to actuators and reading values from the sensors. The decisions of the PLC are relatively high, the actuators receive simple commands from the PLC. For example, the PLC sends an on-off commands to electric engines and the internal controllers of the engine take care about voltage and current (Figure 3.1). PLC decides on actual or seen sensor values and older computed values stored in the local memory. In our model (Figure 3.1), there is no restriction on the internal actuator and sensor logic. The internal logic can see every command and manipulates every physical magnitude. The whole actuator logic is one single sequential circuit and participate in the downstream of the PLC as a part of CPS. So the actuator logic is able to control all actuators at once. We model the same for the sensors. The raw sensor values can be preprocessed by a logic circuit. For example, imagine a presence detector for workpieces, like in the introductory examples, is realised as light barrier with a photoresistor. If a workpiece blocks the light cone into the photoresistor, the resistance increases and the current drops. The PLC only receives a boolean input variable if the light cone is blocked. In the preprocessing, we need an analog-digital converter and a comparison unit to check if the incoming light is under a threshold. The internal logic of the sensors is a single sequential circuit, the same as with actuators. Hence, we are able to model sensor fusion in the internal logic.

Actuator

Sensors

Abstraction

We observe the equivalences of our model. Figure 3.2 shows two communication cycle of the whole system. The sensors read the physical magnitudes from the continuous world and the logic process these measurement at discrete moments in sensor logic. The sensor logic provides the data to the PLC over bus system. The cycles of the sensor and actuators circuits and the PLC does not need to be synchronized. The PLC retrieves the last value on the bus system. After processing the PLC puts the new commands on the bus system and actuator logic tries to change physical magnitudes via the actuators. Every communication and computation need an amount of time, hence the whole system has latency. We don't

consider the latency further.

Functional Description

Every subsystem of our model has a internal state, input and output values. In the following we denote every subsystem with a function. With the use of function we encapsulate the local state of sub system from the other sub systems. Additionally the use of functions implies, that the sensors and actuators does not have uncertainty or inaccuracies, because for every input exists only possible value. Later, the uncertainty can be introduced by relaxing the functions to relations.

Definition 3.1 (Cyber-physical Model)

$$S_t \colon \mathbb{R}^p \to \mathbb{N}^r \tag{3.1}$$
$$SL_t \colon \mathbb{N}^r \to (\mathbb{N} \cup \mathbb{B})^m \tag{3.2}$$
$$PLC_t \colon (\mathbb{N} \cup \mathbb{B})^m \to (\mathbb{N} \cup \mathbb{B})^k \tag{3.3}$$
$$AL_t \colon (\mathbb{N} \cup \mathbb{B})^k \to \mathbb{R}^o \tag{3.4}$$
$$A_t \colon \mathbb{R}^o \to \mathbb{R}^p \tag{3.5}$$

S_t describes the sensors, SL_t the downstream logic of the sensors, PLC_t the programmable logic controller, AL_t the actuator logic circuit and A_t the changes of the physical magnitudes. Every function can vary over time and hold a local hidden state. The cardinality or sets in the Definition 3.1 are not important, as long as the signatures are compatible. So we can allow S_t to return values after a floating point convention, as long as SL_t can handle this type.

We often refer to a physical model of the input. With the equation from Definition 3.1 we are able to describes this term more precisely.

Definition 3.2 (Physical Input Model)

$$pim_{t+1} =_{\text{def}} SL_t \circ S_t \circ A_t \circ AL_t \tag{3.6}$$
$$: (\mathbb{N} \cup \mathbb{B})^n \to (\mathbb{N} \cup \mathbb{B})^m. \tag{3.7}$$

The physical input model is the counterpart of the PLC. The input model takes PLC's output and transform this into input for the next turn.

Equivalence is the claim for the same behaviour under the same input. In our model (Figures 3.1 and 3.2) we have multiple level of inputs and outputs. For example, we can demand equivalence on the observable physical magnitude under the same PLC input. This excludes the sensors and sensor logic from the proof. It is important to state, that you can grab input and output values from every connection in the model. This is useful for testing custom boundaries of submodules in the model. Additionally it can be necessary to replace some sub modules with stub implementations. With this technique we can relax equivalence. Moreover you can plug in different kinds of input models, beginning with non-deterministically choice of every possible input value and ending with a full correct physical input model (Definition 3.2) including user interaction. We address the input model in Theorem 3.7.

3.2 Equivalence of Programs

We have defined our environment, in which the software is embedded. In this section, we compare two software revision for the same PLC system. Imagine, we make a bisimulation

with the same environment, but with two PLC with different software revisions. We would describe the equivalence of both programs a, b as the equivalence of the output.

$$\forall x \in \mathbb{I}\colon f_a(x) = f_b(x). \tag{3.8}$$

Equivalence Assumption

In Equation (3.8) there are many implicit assumptions. The output space $\mathbb{O}_a$ and $\mathbb{O}_b$ of both programs must be compatible in respect to the equivalence relation $=$. The input space $\mathbb{I}$ and the input value x are identical between both function. Both programs must be expressible as pure function $f\colon \mathbb{I} \to \mathbb{O}$, with no internal state. The ST_0 programs has internal memory, that need to store between the function applications. Of course we can concatenate the ST_0 program nth times, hence the state keeps local in the first n turns. But we test only the output equivalence after nth turn. Starting with $n = 0$ and an iterative increasing n lead us to bounded model checking. For stronger statement we let n go to infinity, and so we need to talk about infinite input and output vectors.

Infinite Sequences

In the rest of this section, we extend Equation (3.8) to Definition 3.3. In the following we talk over infinite sequences, similar to ω-structures. We denote a infinite sequences $\bar{s}$ with a top vertical line, and A^ω denotes the set of all possible infinite sequences with elements of A:

$$A^\omega =_{\mathrm{def}} \underbrace{A \times A \times \cdots \times A}_{\infty} \tag{3.9}$$

Perfect Equivalence

We need this notation for the set of possible input vectors $\mathbb{I}$ and output vector $\mathbb{O}$. Equation (3.8) becomes suitable if we use infinite input sequences $\bar{x} \in \mathbb{I}^\omega$.

$$\forall \bar{x} \in \mathbb{I}^\omega\colon \bar{f}_a(\bar{x}) = \bar{f}_b(\bar{x}) \tag{3.10}$$

We define the functions $\bar{f}_{a,b}$ with $\bar{f}_{a,b}\colon \mathbb{I}^\omega \to \mathbb{O}^\omega$ to return the corresponding infinite output trace of $\bar{x}$. Equation (3.10) is writable as a safety condition in linear temporal logic (LTL) [BK08]:

$$\mathbf{G}(o_a = o_b) \tag{3.11}$$

o_a denotes the actual output from program a (resp. o_b). Note, LTL notion hides several aspect of the equivalence, for example the input, but gives a clearer view on the temporal constraints.

Conditional Equivalence

The above considerations are fine if the two programs a, b are *perfectly equivalent*. Moreover we have not addressed the problem, that new revision introduces intentionally different behaviour, for example for fixing a erroneous program behaviour. We relax the perfect equivalence in Equation (3.11) by introducing a premise p, which disables the equality claim on intentional changed behaviour. We can declare p to excludes certain traces or states, for example the trace with the bug fix.

$$\mathbf{G}(p \to o_1 = o_b) \tag{3.12}$$

We state the equivalence more explicitly in first-order logic (FOL). But keep in mind, that a lot of equivalence claims after Definition 3.3 can be expressed in LTL with some extension on the signature.

Definition 3.3 (Equivalence of Programs) *Let a, b two programs, than a is equal b to a relaxation predicate ϕ and the input spaces $\mathbb{I}_a$ and $\mathbb{I}_b$:*

$$a \approxeq_\phi b \text{ iff } \forall\ \bar{i}_a \in \mathbb{I}_a^\omega,\ \bar{i}_b \in \mathbb{I}_b^\omega: \quad \forall t \geqslant 0: \quad \phi(\bar{i}_a, \bar{i}_b, \bar{f}_a(\bar{i}_a), \bar{f}_b(\bar{i}_b), t) \rightarrow f_a(\bar{i}_a) \simeq_t f_b(\bar{i}_b) \tag{3.13}$$

where $\mathbb{I}_{a,b}^\omega$ denote the set of all infinite input sequences for program a, resp. b, $\bar{f}_a$ the output sequences from a (resp. b) and $\simeq_t \subseteq \mathbb{O}_a \times \mathbb{O}_b$ be equivalence relation on the output space depending on the turn (time) t.

In Equation (3.12) we quantify implicitly over both input spaces $\mathbb{I}_{a,b}^\omega$ and the turn counter t. The output from the current step $f_{a,b}(\bar{i}_{a,b})[t]$ depends on the program state and the input $\bar{i}_{a,b}$ seen until turn t. Of course, the Definition 3.3 is not causal[1] and for the implementation with invariants, we restrict the sequence of input and output to the past and actual values.

The premise p becomes $\phi(\cdot)$, a predicate over the both input $\bar{i}_{a,b}$, both output sequences $\bar{f}_{a,b}$ and the current turn t. $\phi(\cdot)$ has the same role as p in Equation (3.12), it relaxes the equivalence. See Section 3.4 for possible conditions on SMV level. Moreover, a LTL formula for $\phi(\cdot)$ is often enough expressive and can be translated into FOL. With the old output values $\phi(\cdot)$ is able to restrict the equivalence claim to physically and behavioural correct inputs of the PLC. For example, imagine our introductory example from Figure 1.2. If w_1 is valid and the conveyor runs, then w_2 will be valid eventually. This can be expressed more formally as a LTL formula:

$\phi(\cdot)$

$$\mathbf{G}(w_1 \rightarrow w_1\, \mathbf{W}(run\, \mathbf{U}\, w_2)) \tag{3.14}$$

where $\mathbf{W}$ is the weak until operator [BK08; Lam02]

$$a\, \mathbf{W}\, b =_{\text{def}} a\, \mathbf{U}\, b \vee \mathbf{G}\, a. \tag{3.15}$$

Equation (3.14) claims, that if w_1 recognizes a workpiece, and run keeps true, the workpiece will arrive at w_2. If a software revision starts and stops the conveyor belt between the transportation process form w_1 to w_2, the input model is no longer a valid abstraction. $\phi(\cdot)$ has to handle disparity in input and output sequences from a, b itself. In this thesis we often assume, that the common variables of both programs a, b are equal. Above, we draw a picture of bisimulation of two PLC in the same environment, if the input spaces of both revision are unequal $\mathbb{I}_a \neq \mathbb{I}_b$, we have to adapt the input vector. Often, the new revision just needs new sensor values, a projection is enough to derive the smaller input vector. With increasing complexity of $\phi(\cdot)$, we need more to argue about the proved equivalence statement and his soundness.

Another relaxation is the equivalence relation $\simeq$. A simple symbolic-wise equivalence of both output vector is sufficient in simple cases. But often a new revision introduces output variables for new actuators or removes output values, the relation $\simeq$ need to map between both output vectors. For example we can define $\simeq$ by using to projection functions $\pi_{a,b}$, that maps the two output spaces $\mathbb{O}_{a,b}$ into a common space $\mathbb{O}_{ab}$:

$\simeq$

$$(a_1, \ldots, a_j) \simeq_t (b_1, \ldots, b_k) \text{ iff } \pi_a(a_1, \ldots, a_j) = \pi_b(b_1, \ldots, b_k). \tag{3.16}$$

[1] In system theory, a causal system does not know his future inputs.

We can see $\pi_{a,b}$ as a selection of arbitrary position from the given vector. Moreover $\simeq_t$ can vary over time and use relations to map single scalars from the output vector. As an example, assume a change in the units of a physical pressure tension actuator using bar instead the previous mmhg:

$$p\colon \text{mmhg} \times \text{bar} \to \mathbb{B} \tag{3.17}$$

$$p(x,y) = y - \epsilon \leqslant 750.061683 \cdot x \leqslant y + \epsilon. \tag{3.18}$$

We use a small ϵ to be aware of rounding errors. For the soundness, we have to consider, why we can assume equivalent input for both PLC, if the output vectors may only partially equivalent. This consideration needs knowledge over the environment, that is controlled by the PLC.

$\phi(\cdot)$ and $\simeq$ should be flexible to be suitable for wide range of equivalence statements, so we make less restriction to the predicates. Later we have to ensure, that the predicates can be expressed in the model checker (Section 3.4) and we need to consider the soundness of the equivalence with respect of the both relaxation.

For our later use the following equivalence relation is sufficient.

Definition 3.4 (Equality of time steps) *$\simeq$ is a predicate $\mathbb{O}_a \times \mathbb{O}_b \to \mathbb{B}$, that decide over the equality from the program outputs of one turn.*

Equivalence Templates

We derive from Definition 3.3 various simpler cases of equivalence, which we want to label. We begin with the easiest one, the equivalence in every step under unrestricted and equal input $\phi = \bar{i}_a = \bar{i}_b$ and no relaxation on the outputs.

Definition 3.5 (Perfect Equivalence) *Two programs a, b are* perfect *equivalent $a \approxeq^p b$ iff*

$$\forall \bar{i} \in \mathbb{I}^\omega\colon \forall t \geqslant 0\colon \bar{f}_a(\bar{i})[t] = \bar{f}_b(\bar{i})[t] \tag{3.19}$$

or as LTL formulae:

$$\mathbf{G}(\bar{f}_a(\bar{i})[t] = \bar{f}_b(\bar{i})[t]) \tag{3.20}$$

The inputs are equal, because the output of both PLCs are equal, and we assume that there is no other possible influence beside the PLC that modifies the system. Additionally, we often talk about perfect equivalence if $\simeq$ is only a projection to the common variables and we can assure the input equivalence.

If we exclude certain inputs or program states from the equivalence, we retrieve the conditional equivalence.

Definition 3.6 (Conditional equivalence) *Two programs a, b are* conditional *equivalent $a \approxeq^c b$ iff*

$$\forall \bar{i} \in \mathbb{I}^\omega\colon \forall t \geqslant 0\colon \phi(\bar{i}[t],\ \bar{f}_a(\bar{i})[t],\ \bar{f}_b(\bar{i})[t]) \to \bar{f}_a(\bar{i})[t] \simeq^\pi \bar{f}_b(\bar{i})[t] \tag{3.21}$$

where $\simeq^\pi$ only uses projections.

The conditional equivalence comes often in two fashions. One fashion is to excludes all ω-structures if a certain input occurs.

$$\mathbf{G}(\ p(\bar{i}) \rightarrow f_a(\bar{i})[t] = f_b(\bar{i})[t]\ ), \tag{3.22}$$

or we can state a stronger version with the until operator:

$$(\ f_a(\bar{i})[t] = f_b(\bar{i})[t]\ )\ \mathbf{U}\ p(\bar{i}[t]) \quad . \tag{3.23}$$

Equation (3.23) includes equivalence for turns until the bad input happens.

Definition 3.3 allow more complex statements. For example we could disable the equivalence for certain amount of steps. We could relax the equivalence in time by claiming

$$\begin{aligned} \forall \bar{i} \in \mathbb{I}^\omega \colon \forall t \geqslant 0 \colon \quad & f_a(\bar{i})[t-1] \simeq f_b(\bar{i})[t] && (3.24)\\ \vee\ & f_a(\bar{i})[t-1] \simeq f_b(\bar{i})[t-1] && (3.25)\\ \vee\ & f_a(\bar{i})[t-1] \simeq f_b(\bar{i})[t-2] \quad . && (3.26) \end{aligned}$$

All equivalence statements, that doesn't use a strong equivalence on the output and assumes equality on the input variables, harbors a problem. This problem already occurs with projection in $\simeq$. The output from the both PLC may not be equal in the previous turns, but we still assume that the both environment in the bisimulation behave the same and give the PLCs the same input values. This assumption needs justification for keeping the proof sound. We make justification for our proofs within the case study, that bases on the separation of the influence between the not common output variables and the input variables. Such a justification can be done automatically if an appropriate model of the environment exists.

Models of the input space are costly to create. The input values at step t depend on input, output and internal from the past. Additionally we need to model random events, like errors or human interventions. It is easier not assume a specific input and allow every possible value in an time step.

Theorem 3.7 (Restriction on input values) *Let a, b be programs and $\phi_P(\cdot)$ a falsifiable condition, for example a physical input model. It holds*

$$a \approxeq_{\phi_P} b \nrightarrow a \approxeq_\top b \qquad\qquad a \approxeq_{\phi_P} b \nleftarrow a \approxeq_\top b \tag{3.27}$$

Proof 3.8 (Theorem 3.7) *"$\leftarrow$" If a, b are equal on all possible input, then they are equal in the subset of input restricted by ϕ_P.*

"$\nrightarrow$" Assume $\phi_P(\cdot) := \bot$ and a and b are not equivalence ($a \not\approxeq_\top b$) but $a \approxeq_{\phi_P} b$ holds. ■

We learn from Theorem 3.7, that $\phi(\cdot) = \top$ is the strongest proof in respect to the input values. If we can proof the equivalence with all possible input values, the equivalence holds in all input models. On the other side if we can not prove the equivalence with $\phi(\cdot) = \top$, then the equivalence can still be valid in the real world. The equivalence $a \approxeq b$ is transitive under assumption, that $\simeq$ is a functional relation.

Theorem 3.9 (Transitive equivalence) *Let a, b, c be programs with $a \approx_\phi b$ and $b \approx_\psi c$, where the used $\simeq$ is a functional relation.*

$$\forall x \in \mathbb{O}_a \colon \exists y, z \in \mathbb{O}_b \colon x \simeq y \wedge x \simeq z \rightarrow y \simeq z. \tag{3.28}$$

then

$$a \approx_{\phi \wedge \psi} c \tag{3.29}$$

holds.

Proof 3.10 (Theorem 3.9) *We retrieve $\simeq_t^{a,b}$ and $\simeq_t^{b,c}$ after the assumption.*

We choose a equivalence relation $\simeq_t^{a,c}$

$$x \simeq_t^{a,c} z =_{\text{def}} \exists y \in \mathbb{O}_b \colon x \simeq^{a,b} y \wedge y \simeq^{b,c} z \tag{3.30}$$

We know, that such $y \in \mathbb{O}_b$ exists from the functional assumption of both $\simeq$ if we restrict the equivalence to $\phi \wedge \psi$.■

3.3 Generating SMV models

We transfer single ST_0 programs into the SMV (Symbolic Model Verifier) input format. The ST_0 compute one scan cycle and one cycle becomes on transition in the model. Every ST_0 program becomes a SMV module, parameterized with input values from the sensors and providing the output values.

An example of the introductory example is given in Appendix A.2.

3.3.1 SMV and IC3

Symbolic Model Verifier is a symbolic model checker (SMC) which represents the states in a symbolic manner with binary decision diagrams (BDD) [Bur+92]. A BDD describes a whole set of states. In contrast to traditional explicit model checking, where every state is stored explicitly.

SMC We cite [SB11]. A finite state system, $S : (\bar{i}, \bar{x}, I(\bar{x}), T(\bar{i}, \bar{x}f, \bar{x}'))$, consists of inputs $\bar{i}$, states variables $\bar{x}$, initial states described by $I(\bar{x})$ and a propositional formula $T(\bar{i}, \bar{x}, \bar{x}')$ for describing the transition from states $\bar{x}$ to states $\bar{x}'$. If we want to prove, that all reachable states $R(\bar{x})$ from the initial states satisfied a safety property $P(\bar{x})$, we write

$$R(\bar{x}) \rightarrow P(\bar{x}). \tag{3.31}$$

The set of reachable states $R(\bar{x})$ is developed inductively by apply the transition relation $T(\bar{i}, \bar{x}, \bar{x}')$ repeatedly until we hit the fix point.

Induction Instead of finding the set of reachable states, we could prove $P(\bar{x})$ by induction over S. First, we prove $P(\bar{x})$ for every initial state $I(\bar{x}) \rightarrow P(\bar{x})$. Second, we show, that taking a transition does not violate the safety property $P(\bar{x}) \wedge T(\bar{i}, \bar{x}, \bar{x}') \rightarrow P(\bar{x}')$ for every input $\bar{i}$. These two steps are called *initiation* and *consecution*.

IC3 $P(\bar{x})$ is seldom a valid an inductional to S. IC3's goal is to find a propositional formulae $F(\bar{x})$, which is a valid invariant of S and $F(\bar{x})$ implies the safety property $P(\bar{x})$ [Bra11; McM03]. The proof obligations are

- $I(\bar{x}) \rightarrow F(\bar{x})$
- $F(\bar{x}) \wedge T(\bar{i}, \bar{x}, \bar{x}') \rightarrow F(\bar{x}')$
- $F(\bar{x}) \rightarrow P(\bar{x})$,

where $\bar{i}, \bar{x}, \bar{x}'$ are all-quantified. The idea is the search for a suitable $F(\bar{x})$. IC3 starts with an over approximation of the reachable states, which are successive tighten by counterexamples from bounded model checker instances. A counterexample can be spurious, caused by the over approximation, or a real counterexample for the safety property $P(\bar{x})$. IC3 is aware of this and guarantees termination with a suitable $F(\bar{x})$ or an valid counterexample.

SMV model

A SMV model contains several modules. A module declares variables with type and initial values as long with the transition between states. For our purpose we need enumeration, boolean and bit vector data types. The enumeration types covers the PLC's enumeration type completely. PLC's signed and unsigned integers variables and values become a bit vector in SMV. We used the nuXmv [Cav+14] model checker, which forces a strict typing with no automatic promotion of variables and values. Hence, we have fixed the bit vector length to an certain size, that is suitable for the case study, but dangerous if the PLC software rely on overflows. A SMV module can be parameter variables from the outer scope. The parameters get substituted at the instantiation of a module. The main module is the entry module of the model checker. We define in the main module the sensor values as SMV input variables (`IVAR`) and inject them into translated modules as module parameters. Input variables choose every possible element of their domain non-deterministically. Hence, every possible combination of input values are considered in the prove. For every state variable can have an assignment for the initial value and the value in the next state. With `init(v) := x`, the variable `v` becomes `x` in the initial state. Whereas `next(v) := expr` defines the value of `v` in the following state. If we do not give an initial or next assignment, the variable becomes undefined.

3.3.2 Symbolic Execution

We use symbolic execution to create the next assignments for every variable in the program body. The init assignments are determined by the rules of the initial values from IEC 61131-3. The program body is in ST_0, hence we need not to consider loops or function calls. The downside are large program bodies. Normal symbolic execution, with creation of new execution state for every branch, leads to unwieldy amount of execution state, caused by the exponential blow up. In the case study, we reach over 13 billion different execution paths for last revision. Instead we use the idea of single static assignment (SSA) with it's Φ nodes [Cyt+89]. As a prerequisite for SSA, every variable can only assigned once. This requirement can easily be ensured with variable renaming and introducing new variables in the program body. Φ nodes introduce the Φ function within the program flow chart. These Φ function works like an if-else-expression:

$$\Phi(cond, A, B) =_{\text{def}} \begin{cases} A & : cond \\ B & : \neg cond \end{cases} \tag{3.32}$$

The example in Figure 3.3 shows a simple code fragment in ST_0. The new value of variable `b` depends on `a` and `c`. After symbolic execution of the first if statement, we get three different states: $a = 1$ as condition and $b := 3$, resp. $a = 2$, $b := 5$ and $a \neq 3 \wedge a \neq 5$, $b := 7$. In SSA we add a Φ node after the if statement (Node 7) and merge the three reached states into one state by expressing b with the Φ-functions:

```
 1  IF a = 1 THEN
 2    b := 3
 3  ELSEIF a = 2 THEN
 4    b := 5
 5  ELSE
 6    b := 7
 7  END_IF;
 8  IF c = 1 THEN
 9    b := b * 2;
10    b := b + 1;
11  ELSE
12    (* empty *)
13  END_IF;
14  b := b + 1
```

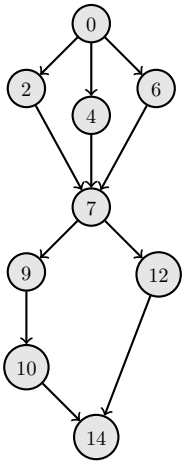


Figure 3.3 Simple ST_0 program fragment with visualized workflow

$$\begin{aligned} b_7 :=& \Phi(a = 1,\ 3 \\ & \Phi(a = 2,\ 5,\ 7)) \quad . \end{aligned} \tag{3.33}$$

We only need to evaluate the next if statement in one state, where $b = b_7$, instead of three different states. If we encounter an already assigned variable on the right hand side, we substitute it, with the Φ expression. Figure 3.4 gives the complete assignment of b after symbolic execution of Figure 3.3. The Φ-function are replace by the `case` expression of SMV. A case expression consists of semicolon separated entries with a guard and a expression. The guard stands on the left of the colon, and determines if the right expression is evaluated and substitutes the case expression. The case expression becomes the value from the first entry, which guard is true. The guard `TRUE` denotes the default entry in a case expression.

In the classical symbolic execution we would have six possible states in which we evaluate line 14. In general this construction is exponential but leads less complexity in guards and expressions. SSA achieves linear time with the cost of more complex formula. In Node 14 b is

$$\begin{aligned} b_{14} :=& \Phi(c = 1, \\ & \Phi(a = 1,\ 3, \\ & \quad \Phi(a = 2,\ 5,\ 7)) * 2 + 1, \\ & \Phi(a = 1,\ 3, \\ & \quad \Phi(a = 2,\ 5,\ 7))) + 1. \end{aligned} \tag{3.34}$$

Figure 3.5 gives the algorithm for generating the expression of variable assignments from a statement. We iteratively apply the algorithm on the program body. If a variable is not assigned in a scan cycle, it keeps his value. Hence, we start the iteration with an identity variable assignment

$$state : var \mapsto var \quad . \tag{3.35}$$

The algorithm needs a symbolic replacement $\mathcal{E}(expr, state)$ for substituting each occurrence of variable in *expr* with the assigned value in *state*. We use template matching for determining the type of the statement and encode if statements $IF(c_0, s_0, \ldots, c_n, s_n)$ as a sequence of guards c_i and list of statements s_i, where the else branch has `TRUE` as his guard.

```
 1  next(b) :=
 2    (case c = 1 :
 3            ((case a = 1 : 3;
 4                  a = 2 : 5;
 5                  TRUE  : 7;
 6              esac) * 2 ) + 1;
 7          TRUE  :
 8             (case a = 1 : 3;
 9                  a = 2 : 5;
10                  TRUE  : 7;
11              esac);
12     esac) + 1;
```

Figure 3.4 SMV assignment for variable `b` from Figure 3.3

Input: A ST_0-allowed statement *stmt* and a variable mapping
state: $L(\langle name \rangle) \to L(\langle expr \rangle)$.
Output: The updated variable map *currentState*.
begin
  currentState := *state*;
  if *stmt matches* $IF(c_0, s_0, \ldots, c_n, s_n)$ **then**
    for $0 \leqslant i \leqslant n$ **do**
      $state_i$:= copy(*currentState*);
      $d_i := eval(c_i, currentState)$ /* Evaluate c_i in *currentState* */;
      $state_i := SymEx(s_i, state_i)$ /* Update $state_i$ with comparison in d_i */;
    end
    newState := *copy*(*currentState*);
    for *var* ∈ DOMAIN($state_i$) **do**
      /* If *var* is reassigned in $state_i$ */
      if $\exists\ 0 \leqslant i \leqslant n$: $state_i(var) \neq currentState(var)$ **then**
        $newState(var) := \Phi(d_0, state_0(var),$
            $\Phi(d_1, state_1(var), \ldots \Phi(d_{n-1}, state_{n-1}(var), state_n(var))));$
      end
    end
    currentState := *newState*;
  end
  if *stmt matches* $ASSIGN(var, expr)$ **then**
    $currentState(var) := eval(expr, currentState)$;
  end
end

Figure 3.5 $SymEx(stmt, state)$, symbolic execution of ST_0 statements

3.3.3 Optimizations

We present several techniques to reduce the number of variables and bits in our models. Optimizations take place at several levels, during transformation from SFC and ST to ST_0, ST_0 to SMV or in the model checker. We investigate the variables in the SMV model, find differences in SFCs, split proves along the function blocks. Typical for these techniques in regression verification is the exploitation of structure similarities. In a software evolution, we separate the changed from the unchanged software parts to reduce effort for the prove.

Read-only variables

We categories variables in the program in several classes. Variables with write access, are read-only, and we need not to store them into the state space. Instead we express these variables as `DEFINES` in SMV. Input variables of function blocks or functions are substituted during the symbolic execution. These are "written before read" variables overridden before any read happens. In general every "written before read" variable disappears after the symbolic execution. Additionally we remove variables, that have only write access, if they are not used as output variable of the program.

Variable Slicing

We want to partitioning set of variables, into the changed and unchanged variables between software revisions. The common unchanged variables can be shared between the two revision in the SMV model. Figure 3.6 shows to software revisions in ST_0 and SMV. Figure 3.7 gives the computed variable dependency graph.

Let Var_1 and Var_2 be two sets of variables in revision one, resp. two. The set of shared variable $ShVar$ is a subset of both variable domains

$$ShVar \subseteq Var_1 \cap Var_2. \tag{3.36}$$

This definition assumes name equality between both revisions. If this assumption does not hold, we would define a mapping $f\colon Var_1 \rightarrow Var_2$ between the two variable domains. The members of $ShVar$ are determined by search in the dependency graph of variables and the mark of dirty variables.

Definition 3.11 (Variable dependency graph) *A variable dependency graph is a directed graph (V, E) with $V \subseteq (Var_1 \cup \boxdot) \times (Var_2 \cup \boxdot)$ and an edge $e \in E$ is a tuple $e \in V \times V$. An edge $((v_1, v_2), (w_1, w_2)) \in E$ exists only, if the computation of v_1 depends on w_1 or v_2 on w_2.*

Definition 3.12 (Dirty variable) *A node $(a, b) \in V$ of a variable dependency graph (V, E) is in the set of dirty nodes D_V, iff*

- *$a = \boxdot$ or $b = \boxdot$ or*
- *there exists a variable assignment, such that $\mathrm{next}(v_1) \neq \mathrm{next}(v_2)$.*
- *or one of dependent variables are dirty, $\exists s, t \in V\colon s = (a, b) \wedge (s, t) \in E \wedge t \in D_V$*

A variable v is dirty in a given variable dependency graph (V, E), iff $(v, v) \in D_V$

Definition 3.11 defines graph, where the edges denotes dependencies between variables. A vertex $(v_1, v_2) \in V$ is a match of a variable v_1 of the previous version with a variable v_2 of the next version. After definition of *ShVar*, we assume $v_1 = v_2$. If no match exists, either v_1 or v_2 is undefined $\boxdot$. The vertex $(\boxdot, \boxdot)$ does not exists. Every variable w reachable from a variable v is called successor of v, vice versa for predecessor. A node is dirty if one of variables is missing, the next-expression of both variables evaluates different or the node depends on another dirty node. The dirtiness property propagate reverse to the edge direction, also in direction to the predecessor.

Obviously, this technique achieves good savings if the software change effects variables, that have less predecessors. With the variable dependency graph we can show equivalence on syntactical level for every single program output variable.

Lemma 3.13 (Equivalence of clean output variables) *If an program output variable is clean, the variables behaves the same in both revisions.*

Lemma 3.13 follows immediately from the definition of clean variables.

Additionally cleaned variables are at the end of every path in the variable dependency graph. Obviously, we split the graph into a partition of clean and dirty variables. Every input value is clean[2] and at some point all predecessors variables are dirty (Lemma 3.14). Following lemma gives us a reasoning for splitting clean variables into a separated SMV module.

Lemma 3.14 (Dirty End) *Let p a path $(v_1, \ldots, v_k)$, then*

$$\forall\ 1 \leqslant i \leqslant k \colon v_i \notin V_D \rightarrow \forall\ i \leqslant j \leqslant k \colon v_j \notin V_D \tag{3.37}$$

Lemma 3.14 states, if we reach a clean variable in an arbitrary path, then every successor is clean, too. Obviously, this holds by definition, because the dirty flag effects only the predecessors. If a variable is clean, then the node is locally identical in name and next assignment, and every successor is clean. The equality of next assignments is a functional equivalence like Equation (3.8). This equivalence can be solved in different ways. For instance, a simple check for syntactical equivalence gives an over approximation. More effort leads to investigation of the different cases in the expression or modelling this question to SMT or SAT solvers. Checking of the equivalence is an over approximation as long as, we do not know the exact input and state space for the specific variable. We may find a spurious case of inequality in the two next definition, that does not occur at runtime.

With Lemma 3.14 we can argue a simple separation of clean variables from two SMV modules into a new created module. Restate from previous chapter, there is one *main* module, that instantiate the modules for each revision. We introduce a new module with all shared variables *ShVar*, that is instantiated in the top level module. Every *ShVar* in the revision modules becomes a module parameter and uses the instances from the common module of shared variables.

Timer replacement

In this section we try make a timer forwarding for TON timer from Section 2.4.3. TON timers are used to introduce delays into the control flow, by waiting a fix amount of time. TON Timers are function blocks, that depends on the real hardware time and is

[2] In detail, this assumes equality on the sensor values

```
1  FUNCTION_BLOCK  main1
2    VAR_INPUT
3      i : boolean;  END_VAR;
4    VAR a     : boolean;
5        b, c : USINT;
6    END_VAR;
7
8    c := b * 2;
9    IF a THEN b := 1 ELSE b:= 3;
10   a := NOT i;
11
12 END_FUNCTION_BLOCK
```

(a)

```
1  FUNCTION_BLOCK  main2
2    VAR_INPUT
3      i : boolean; END_VAR;
4    VAR a        : boolean;
5        b, c, z : USINT;
6    END_VAR;
7
8    c := z * 2;
9    z := b + 1;
10   IF a THEN b := 1 ELSE b:= 3;
11   a := NOT i;
12 END_FUNCTION_BLOCK
```

(b)

```
1  init(c) := 0;
2  next(c) := b * 2;
3
4
5
6
7  init(b) := 0;
8  next(b) := case   a : 1;
9                TRUE : 3; esac;
10
11 init(a) := FALSE;
12 next(a) := ! i;
```

(c)

```
1  init(c) := 0;
2  next(c) := z * 2;
3
4  init(z) := 0;
5  next(z) := b + 1;
6
7  init(b) := 0;
8  next(b) := case    a : 1;
9                TRUE : 3; esac;
10
11 init(a) := FALSE;
12 next(a) := ! i;
```

(d)

Figure 3.6 Example of common variables in two ST_0 programs. The variables a and b are equal. c is changed. Figure 3.7 shows the graph.

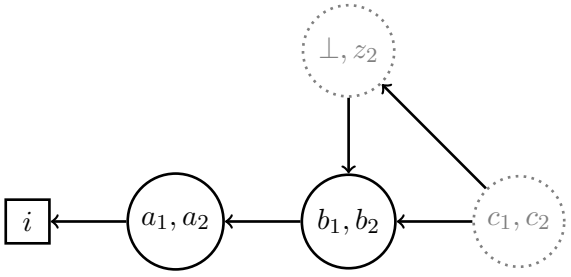
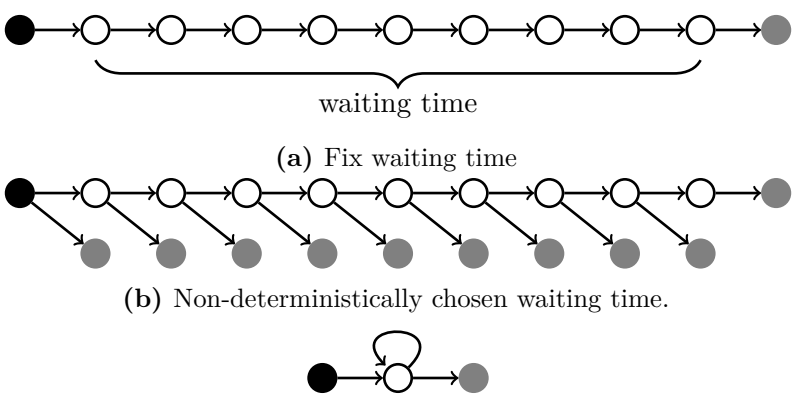


Figure 3.7 Graphical dependency between variables of Figure 3.6. We mark c as dirty, because it depends of a new variable z_2.



(a) Fix waiting time

(b) Non-deterministically chosen waiting time.

(c) Non-deterministically choice if the waiting should end or not.

Figure 3.8 Visualization of timer abstraction. The white marks the waiting nodes. The black node is the entry in the waiting time, and grey the first node after the waiting time.

independent of the function block calls. In Section 2.4.3 we implemented as a counter, that is decremented on every call. We have to ensure that the function block is at most called once in a scan cycle. Others timers may claim different abstraction techniques. We want to fast forward his waiting time.

Restating TON's behaviour, the caller calls an instance of the TON function block, with a waiting time PT and a switch IN. After the call, the caller can retrieve the flag Q from the function block. Q is true, if the IN stayed true over PT seconds.

In our case study, the waiting PT is often a high value, that leads to waiting times and causes disadvantage in the model checking. For the reduction of the waiting time PT we first need to identify the corresponding timers in both software revision. Both timer instances have to be called in the same turns with the same input arguments[3]. Additionally we could claim the equivalence of the internal timer state, because both timers use the same implementation from the preamble. The checks take place in the syntax of the program ST_0 or can be added as a proof-obligation to the model checker. If the assumptions hold we have to possibilities. First we let PT a fix non-deterministically chosen variable, hence the assumption, that we can wait a undefined finite time in both programs (Figure 3.8b). This is a over approximation of one fixed waiting time. We can get rid of the time completely by choosing the Q output non-deterministically. The second approach (Figure 3.8c) is an over approximation of the first one, because it allow infinite waiting times. Figure 3.8 visualize this.

Function block replacing

The abstraction of timers can be extended to abstract whole function blocks. If we find the corresponding function blocks in both revisions and we ensure the same activation in both revisions. Same activation means, that both instances are called at the same turn with the same input arguments. Additionally we need to prove the equivalence of the functions blocks. The function block equivalence is not necessary for timers, because they are defined in the preamble.

If the same activation and function block equivalence holds, we are able to extract the function into a mutual used SMV module. Furthermore, we can make an over approximation of the function block, that returns non-deterministically chose output values.

[3] A relaxation is possible for resets of the timer instances, not considered here

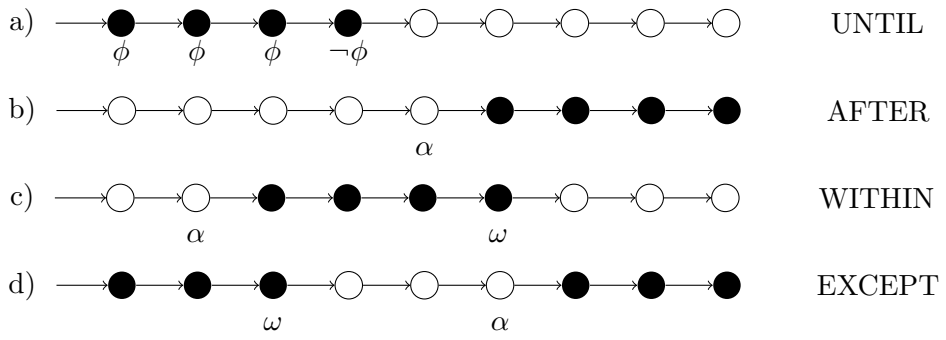


Figure 3.9 Behaviour of the different claims. The black states claims the equivalence, the white states not. a) UNTIL claims equivalence as long as ϕ is true. b) AFTER claims the equivalence, after α was true once. c) WITHIN marks a section of equivalence claim. d) EXCEPT is opposite to WITHIN.

3.4 Equivalence as Invariant

We check the equivalence with IC3, a inductive method for proving invariants, so we need to encode proof obligation for the equivalence Definition 3.3 as an invariant. The main effort is to encode the condition for the equivalence. The relation $\simeq$ often compares the output of both programs from the same turn. Otherwise we store old values in between if needed. We give templates for the encoding of the equivalence in SMV.

Let E be a formulae, that describes equivalence within a turn, E encodes $\simeq$. E compares the same outputs $1 \leqslant i \leqslant n$ from both programs A, B:

$$E := \bigwedge_{1 \leqslant i \leqslant n} E_i, \tag{3.38}$$

where E_i describe the equivalence between two output variable. For a strong equivalence we choose $E_i := A.o_i = B.o_i$. A relaxation is possible by adding functions or making a disjunction, for example

$$E_i := A.o_i = 5 * B.o_i \quad \text{or} \qquad E_i := (A.o_i = B.o_i \vee A.o_i = 0). \tag{3.39}$$

Remember, a relaxation of the equivalence between output variables, requires a investigation if the equality of input variables is sound.

We present four cases of conditional equivalence. We introduce operators, encoded in SMV, on ω-structures for claiming the equivalence in regions of the ω-structure. We call these operators claims, for avoiding confusing with existing LTL operators. Figure 3.9 gives an overview about the claims. In the black states we claim the equivalence E, in the white states E need not to be satisfied. The formula α activates the claim and ω marks the end of the equivalence. The evaluation α and ω has a delay of one step, this can be avoided using next(α) or next(ω). If α becomes true in step t, the equivalence claim begins $t + 1$.

UNTIL The first claim is a derivate of the LTL's weak until operator.

Definition 3.15 (UNTIL claim) $\text{UNTIL}_\phi(\psi)$ *is true iff* ψ *is true as long as* ϕ *is true.*

$\text{UNTIL}_\phi(\psi)$ is defined in LTL as

$$\text{UNTIL}_\phi(\psi) =_{\text{def}} \ \psi \wedge (\mathbf{X}\,\psi \ \mathbf{W} \ \neg\phi). \tag{3.40}$$

The until claim has a slightly different behaviour on the border, than the LTL until operator. The until operator $\psi \,\mathbf{U}\, \phi$ in LTL does not require, that ψ is true, if ϕ is true in the first turn, and if ϕ is true, ψ can be arbitrary. At this point we want to ensure ψ is still valid. This behaviour is inspired from simple exclusion of transition in SFC. We exclude every ω-structure at this point, where ϕ marks the forbidden transitions. The evaluation of transition takes place in turn t, but the excluded steps becomes active in the next $t+1$. Moreover, we evaluate ϕ in turn t and the updated condition is available in $t+1$. The LTL next operator in Definition 3.15 ensures this behaviour on the border. If ϕ never becomes false, the equivalence is claimed over all states.

In SMV we maintain a boolean variable *claim*, that is initialize with true and becomes false in the next turn, if ϕ is false (Figure 3.10). We use this scheme for all defined claims

$$claim \rightarrow E. \tag{3.41}$$

AFTER

The second claim is similar to the negation of the until claim. We claim, that the equivalence E holds after α was true once.

Definition 3.16 (After claim) $\mathrm{AFTER}_\alpha(\phi)$ *is true if and only if* ϕ *is true in every step after* α *was once true.*

We can express $\mathrm{AFTER}_\alpha(\phi)$ in LTL as

$$\mathbf{G}(\alpha \rightarrow \mathbf{G}\,\mathbf{X}\,\phi). \tag{3.42}$$

If α becomes never true, we does not ensure equivalence. The SMV code is similar, but the initialization of the claim bit is false and we use a disjunction (Figure 3.10). The after claim is useful, to skip the initialization of a plant and claim equivalence in normal production cycle.

WITHIN

In the within claim, we want to ensure the equivalence with region on the ω-structure. Formula α detects the beginning of the region and ω the end.

Definition 3.17 (Within claim) $\mathrm{WITHIN}^\omega_\alpha(\phi)$ *is true iff* ϕ *is valid in all states* σ *after* α *and before* ω *iff the following formula holds within an* ω*-structure* $\sigma_0, \sigma_1, \sigma_2, \ldots$.

$$\forall s \in \mathbb{N}: \tag{3.43}$$

$$(\exists t < s\colon (\sigma_t \models \alpha \wedge \quad \forall u > t\colon \sigma_u \not\models \omega)) \rightarrow \quad \sigma_s \models \phi \tag{3.44}$$

Definition 3.17 is equivalent to

$$\mathrm{WITHIN}^\omega_\alpha(\phi) \equiv \mathbf{G}(\alpha \rightarrow \mathbf{X}\,\phi \;\; \mathbf{W} \;\; \omega). \tag{3.45}$$

If α is never true, we do not claim equivalence, otherwise we claim the equivalence until ω becomes true or until infinity. The decoding (Figure 3.10) distinguish between the cases, if we are within or outer the claim. If the claim severe, we wait for α to enter the block. Within the region we wait ω to leave it.

EXCEPT

The opposite of within claim is except claim. It disables the claim within a region of states. The decoding is the same as within, only the initial value for the claim is different.

```
 1  MODULE UNTIL( φ, ψ )
 2  VAR    claim : boolean
 3  ASSIGN init(claim) := TRUE;
 4         next(claim) := claim & φ;
 5  DEFINE INV = claim -> ψ;
 6
 7   AFTER( α )
 8  VAR    claim : boolean;
 9  ASSIGN init(claim) := FALSE;
10         next(claim) := claim | α;
11  DEFINE INV = claim -> ψ;
12
13
14  MODULE WITHIN( α, ω )
15  VAR    claim : boolean;
16  ASSIGN init(claim) := FALSE;
17         next(claim) := case    claim: ! ω;
18                             ! claim:   α;
19                        esac;
20  DEFINE INV = claim -> ψ;
21
22  MODULE EXCEPT( α, ω )
23  VAR    claim : boolean;
24  ASSIGN init(claim) := TRUE;
25         next(claim) := case    claim: ! ω;
26                             ! claim:   α;
27                        esac;
28  DEFINE INV = claim -> ψ;
```

Figure 3.10 Claims encoded in SMV as separate modules. The `claim` variable can be used as a premise, see Equation (3.41)

Definition 3.18 (Except claim) *Let $\sigma_0, \sigma_1, \ldots$ be an ω-structure, then* $\mathrm{EXCEPT}^{\omega}_{\alpha}(\phi)$ *is true iff*

$$\forall s \in \mathbb{N}\colon \sigma_s \models \phi \vee (\exists t < s\colon \sigma_t \models \omega \wedge \neg \exists u\colon t < u < s-1 \wedge \sigma_u \models \alpha) \tag{3.46}$$

Definition 3.18 we can express $\mathrm{EXCEPT}^{\omega}_{\alpha}$ in LTL.

$$\mathrm{EXCEPT}^{\omega}_{\alpha}(\phi) \equiv (\phi \,\mathbf{W}\, \omega) \wedge \mathbf{G}(\alpha \rightarrow \mathbf{X}\, \phi \,\mathbf{W}\, \omega) \tag{3.47}$$

In general we can reduce until to except claim, respective the after claim to within claim, by setting an unsatisfiable α, resp. ω. We grasp these claims for the describing the conditionals needed in the case study in Chapter 4.

3.5 Finding conditions of equivalence

The scenarios in the case study (Chapter 4) show a pattern for finding the condition ϕ for showing the equivalence. We find changes between two SFC (left and right) by traversing both SFC simultaneously, like a bisimulation. We regard difference between both SFC in the condition. The difference can be changed, new or deleted steps, transitions or actions. This was the ways, how we derived manually the conditions for the case study. We want to algorithmize our observation. We need similar structures like Definition 3.11. First, let every reached node in the simultaneously traversing of left and right SFC be called a twin step.

Definition 3.19 (Twin Step (TS)) *Let* $\mathcal{S}_l =_{\text{def}} (S_l, s_{0,l}, T_l, a_l, \sqsubset_l, \prec_l)$ *and* $\mathcal{S}_r =_{\text{def}} (S_r, s_{0,r}, T_r, a_r, \sqsubset_r, \prec_r)$ *be two SFC, then a twin state of* $\mathcal{S}_l$ *and* $\mathcal{S}_r$ *is a tuple* $(l, r) \in S_l \times S_l$.

Note, every twin step has entry, active and exit action from the left and right SFC. We create, during the traversing of the left and right SFC, a directed graph of twin steps and twin transitions between them.

Definition 3.20 (Twin Step Graph (TSG)) *A Twin Step Graph (TSG) is a graph* (V, E), *where* $V =_{\text{def}} (S_1 \cup \boxdot) \times (S_s \cup \boxdot)$ *and* $E = V \times V$.

An edge $e \in E$ *exists, iff there exists a transition in one of the SFC. Every edge* e *is mapped with two guards* $\gamma_l(e), \gamma_r(e)$.

$\boxdot$ denotes a non-matched step. We denote an edge as a twin transition referring to transition in SFC. There are several ways to generate a TSG. For example following is imaginable:

- Simultaneously traversing both SFC and try to match steps and transition in traversing order.
- Matching of steps with respect to step name, without referring to position and transition.

We consider the first approach. Figure 3.12 shows the algorithm for traversing both SFC. The algorithm is similar to breadth-first search. A *queue* holds twin states that needed to be explored and *visited* prevented loops. The main part of Figure 3.12 is the matching between the outgoing transitions, by the comparison $(g_l, l') \approx (g_r, r')$. The relation $\approx$ use in our implementation the guard or name of the target step:

$$(g_l, l') \approx (g_r, r') \;=_{\text{def}}\; g_l = g_r \;\vee\; \text{name}(l') = \text{name}(r') \tag{3.48}$$

Figure 3.11 shows the TSG for the introductory example from Section 1.1 and figure 2.17 with revision I and IIa. The new branch in revision IIa is part of the TSG. The left side (revision I) in this branch is completely undefined ($\boxdot$). The TSG is good base to give algorithms of finding ϕ for proving the equivalence. We see directly, that a until claim with $\phi = \neg(w_2 \wedge d)$ would resolve in equivalent behaviour.

Unequal Transitions

We give to algorithms. The first one targets twin transition with unequal guards. Let $t = (\,(l, r), (l', r')\,)$ be a twin transition from (l, r) to (l', r') with unequal guards $g_l \neq g_r$. Then we conjoin following formulae to the condition

$$(state_l = l \;\wedge\; state_r = r) \rightarrow\; (g_l \leftrightarrow g_r), \tag{3.49}$$

where $state_l$ is the SMV state variable the corresponding left SFC (resp. $state_r$). The formula tie g_l and g_r together. If we take the transition in the left SFC, then we need to take appropriate transition in the right SFC and vice versa. The created formula can be inconsistent and unsatisfiable. This may depend on logical environment, too.

Unequal States

Our second algorithm addresses differences in branches, like in Figure 3.11. Let $t =$

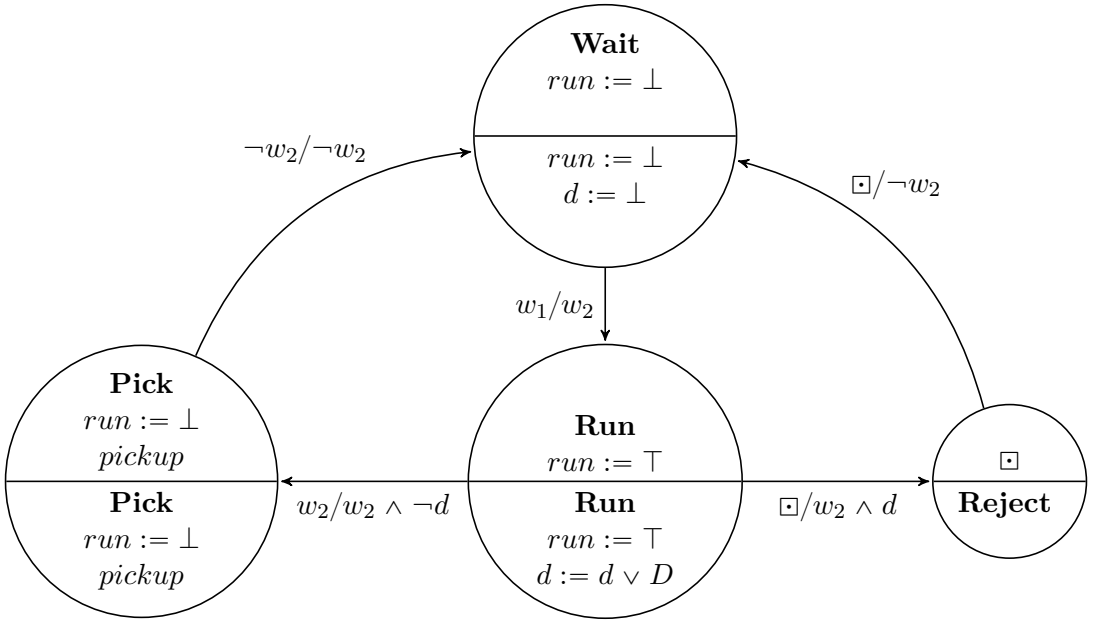


Figure 3.11 Twin State Graph of the introductory example between revision I and IIa from Section 1.1 and SFC from Figure 2.17

$((l, r), (l', r'))$ be a twin transition where the left guard $g_l = \boxdot$ is undefined (w.l.g.). We conjoin the following formulae to the condition, for excluding this transition

$$(state_l = l \ \wedge \ state_r = r) \rightarrow \neg g_r. \tag{3.50}$$

Enrich TSG

Proceeding, the TSG is a help for humans for given a equivalence condition. The graph can be enriched with more information, for example about assumption of the variable values or environmental states. We can try to determine the value for every variable in each step, by symbolic execution of the SFC. It is possible, that not every variable reaches a fixed point. For example, loops in SFC are allowed and counting of these loop would be unbound in general. In our example Figure 3.11 this approach would find the variable assignments for the *Reject* step.

Symbolic Execution

Guarantees

Another extension for TSG is an approach of guarantees. We want to gather information from the transition guards into the local step. An example explains the intention. Figure 3.13 shows a simple state automata, where the nodes G, H and I was introduced in a revision. For example, the author detected a waiting time in step B and decided to use the waiting for other operations (G, H, I). If we can prove, that the relevant parts of the SFC state and of the environment are equal after leaving step B and step I, we can consider the effects of the changes as local. Hence, we can take the branch in the bisimulation and argue, that the input is still equal in step C. The relevant parts covers knowledge from the environment. We have to known which influence the actuator values and sensor values have to each other.

Previously, we execute the actions of the steps symbolic. Now, we consider the transition guards of the SFC. Both are coupled together, because the actions set the actuator values and the transition guards checks the sensor values. We want to carry the information of the sensor values given by satisfying a guards as long as possible to the successor steps.

But if a guard holds in a scan cycle, it does not need to hold in the next scan cycles. A guarantee describes the knowledge, that sensor value holds until an event, like certain actuator values, occur. Such a guarantee is revoked under circumstances described by an propositional formula. Back to our introductory example Figures 2.17 and 3.11, we can

Input: Two SFC$_0$ $\mathcal{S}_l = (S_l, s_{0,l}, T_l, a_l, \sqsubset_l, \prec_l)$ and $\mathcal{S}_r = (S_r, s_{0,r}, T_r, a_r, \sqsubset_r, \prec_r)$
Output: A twin state graph (V, E) with γ_l, γ_r as mapping from edge to left and right guard

begin
  $visited := \{\}$;
  $queue := \{\ (s_{0,l}, s_{0,l})\ \}$;
  while $queue \neq \varnothing$ **do**
    $(l, r) := \text{poll}(queue)$;
    $suc_l := \{(g, l') \mid \forall (l, g, l') \in T_l\}$;
    $suc_r := \{(g, r') \mid \forall (r, g, r') \in T_r\}$;
    $suc := \{\}$;
    if $|suc_l| = 1 \wedge |suc_r$ **then**
      $\{(g_l, l')\} := suc_l$;
      $\{(g_r, r')\} := suc_r$;
      $suc := \{\ (g_l, g_r, l', r')\ \}$
    else
      for $(g_l, l') \in suc_l$ **do**
        for $(g_r, r') \in suc_r$ **do**
          if $(g_l, l') \approx (g_r, r')$ **then**
            $suc+ = (g_l, g_r, l', r')$;
            Remove (g_l, l') from suc_l;
            Remove (g_r, r') from suc_r;
          end
        end
      end
    end
    for $(g_l, g_r, l', r') \in suc$ **do**
      $e := (\ (l, r),\ (l', r')\ )$;
      $E = E \cup \{\ e\ \}$;
      $\gamma_l(e) := g_l$;
      $\gamma_r(e) := g_r$;
      $queue = queue \cup \{(l', r')\}$;
    end
  end
end

Figure 3.12 Creating of TSG by simultaneously traversing

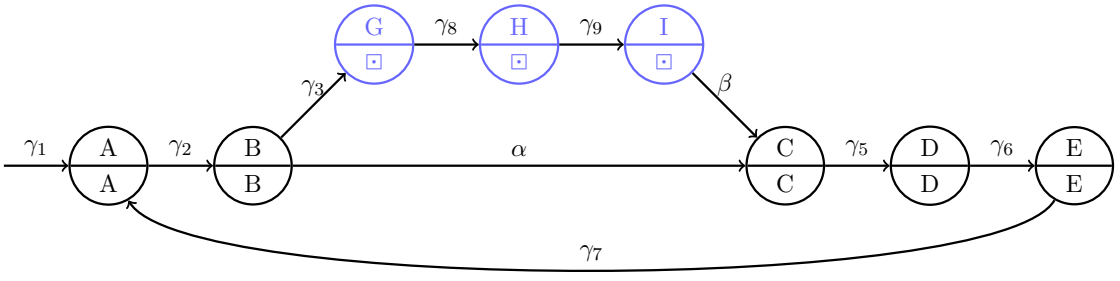


Figure 3.13 Guarantee example with a sketch of a TSG. G, H, I are introduced to bypass a waiting time between B and C.

easily give the guarantees for the sensor values w_1 and w_2. We start with a definition in LTL.

$$\neg run \rightarrow w_1 = \mathbf{X}\, w_1 \tag{3.51}$$
$$\neg run \rightarrow w_2 = \mathbf{X}\, w_2. \tag{3.52}$$

Equation (3.51) connects the actuator command run with the immutability of sensor values w_1 and w_2. The sensors can only changed if conveyor belt moves, regardless of human interaction.

We can propagate the sensor values from the guards to the states with these guarantees (Figure 3.11):

$$\mathcal{G}(Wait) := \neg w_2 \wedge \neg w_1 \tag{3.53}$$
$$\mathcal{G}(Run) := \boxdot \tag{3.54}$$
$$\mathcal{G}(Reject) := \boxdot \tag{3.55}$$
$$\mathcal{G}(Pick) := w_2 \wedge \neg w_1 \tag{3.56}$$

The guarantee approach need a model, which sensor value is constant under some conditions. Normally, the physical input model *pim* (Definition 3.2) contains this information. A proof for guarantee with the *pim* needs to consider, that the physical input model has a internal state, too.

For some cases, a physical model would be too much effort. As end note of this section, we state an simpler restriction. There is similarity between the guarantees and default logic. Default logic offers a way to apply rules, if nothing is in conflict with them. We say, that the sensor value does not changed until we have conflict caused by an actuator. For a reminder, default logic [Wik13] uses rules in the manner $p_1, \ldots, p_n$ implies $q_1, \ldots, q_n$ as long as the justification $j_1, \ldots, j_n$ is consistent. The rule can applied if the justification evaluates to false or is unknown. The justification makes the default non-monotonic as it might prohibit to apply the rule if more new knowledge is available. The rule is written as follows:

$$\frac{p_1, \ldots, p_n : j_1, \ldots, j_n}{q_1, \ldots, q_n}. \tag{3.57}$$

We can state our Equations (3.51) and (3.52) in default logic Equation (3.57):

$$\frac{w_1 : run \vee \neg w_1'}{w_1'} \qquad \frac{\neg w_1 : run \vee w_1'}{\neg w_1'} \tag{3.58}$$
$$\frac{w_2 : run \vee \neg w_2'}{w_2'} \qquad \frac{\neg w_2 : run \vee w_2'}{w_2'}. \tag{3.59}$$

You can read the first rule: w_1 implies w_1 in the next turn is true, as long run is not true[4] or contrary is known about w_1'. If no above rules apply, $w_{1,2}'$ are arbitrary.

[4] false or unknown

4. Case Study

[VH+14] describes the *Pick and Place Unit*[1] (PPU) case study. The main focus of the case study is the software engineering within the evolution of software, hardware and requirements (Figure 4.1) and not (regression) verification. We make some adaption in the source code to lower the complexity for the equivalence proves. There is one kind of change, that causes inequalities and brings complexity.

Drift

The developers of the case study introduced empty steps in SFCs. These empty step have no assigned actions or outgoing transition guard differing from `TRUE`. This step introduce a delay of one scan cycle for the SFC. The difference is not observable for a human in the running plant, because the scan cycle takes four milliseconds in the case study. If the empty step is within a loop in the SFC, the delay accumulates in each iteration, hence the delay is not fixed. We call this behaviour a drift, then both simulations drifts apart and are not in synchronized. There are two simple solutions: Introducing a delay step in the old revision, or removing the delay step in the new revision. We choose the first approach, after consulting the developers of the case study. Another often change is the bringing forward of actions, in which certain actions, especially assignments, are executed some steps earlier in the new revision than in the old revision. The effect is the same as for drifts, but only for some variables. For solving this problem, we can relax the equivalence for these affected variables in certain time windows. Like every equivalence weakening this requires considerations about the soundness and the assumption input variables equivalence. To apply this solution on drifts, we need to argue why we can resynchronize both simulations at a certain point of time.

Outline

In the rest of this section we introduce the PPU with it's components and process workflow. In Section 4.1 we give the differences of hardware and software between the revision, as well as explanation of the equivalence condition and source code adaptions. In Section 4.2 we represents runtime of nuXmv proofing the equivalence.

Not every revision in Figure 4.1 impacts the PLC software changes. The considered scenarios are in the central ladder, excluding scenarios are 4a, 4b and 6 in Figure 4.1.

PPU

The PPU consists of two to five SFCs, depending on the scenario, and one program *Main*. The PLC triggers the program every four milliseconds in all scenarios. Table 4.1 gives an overview when introduction and changes on the SFC function blocks. We distinguish between changes, that intentionally introduce new behaviour (N) in SFC, other changes

[1] `https://www.ais.mw.tum.de/ppu/`

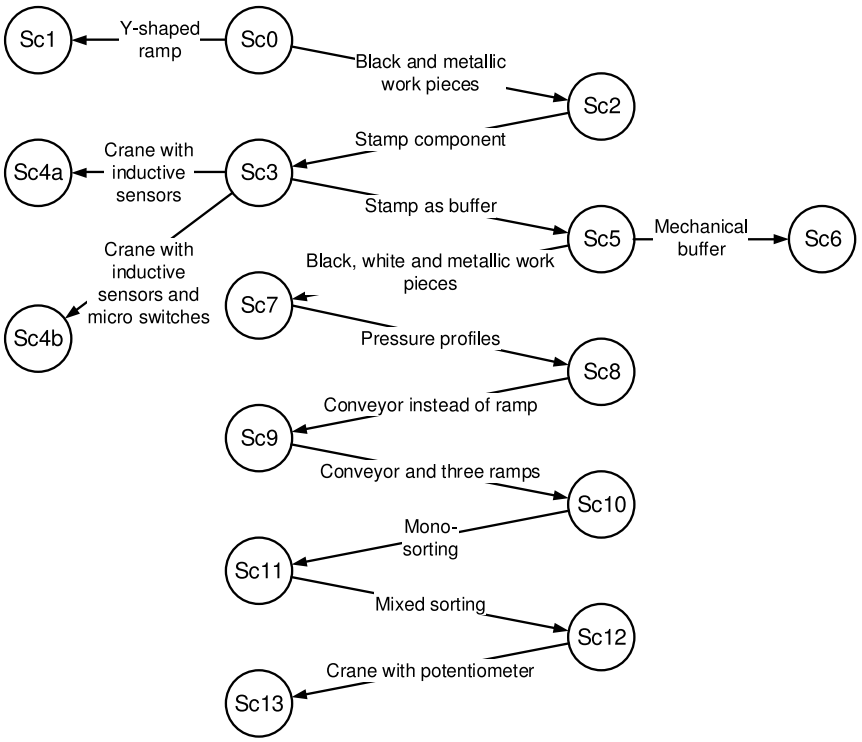


Figure 4.1 Scenarios of the case study from [VH+14] with the major change and association. Every scenario step includes a software change, except scenario 4a, 4b and 6.

with effects on equivalence (C), without effects (∅) or only addition new sensors or actuator variables (NS). If the behaviour is not comparable between revision, we set the X mark and = denotes equivalent function blocks. This comparison bases on the syntax of the function blocks and program.

Hardware In the center of the PPU is a crane (corresponds to the Crane function block), that transports workpieces between magazine, stamp and conveyor belt (Figure 4.2c). The stack (function block Magazine) provides fresh workpieces. The stamp imprints a logo on workpieces and corresponds to Stamp function block. The conveyor belt, introduced in

Scenario	Main	Crane	Magazine	Stamp	Conveyor	Pusher
Sc0	I	I	I			
Sc1	=	=	∅			
Sc2	NS	N	∅			
Sc3	=	=	=	I		
Sc5	∅	N	=	G		
Sc7	NS	N	=	=		
Sc8	∅	N	=	=		
Sc9	C	N	=	=	I	
Sc10	C	∅	=	=	N	I
Sc11	NS	∅	=	=	∅	X
Sc12	=	∅	=	=	=	X
Sc13	C	G	=	=	=	=

Table 4.1 Function block changes during the evolution. Symbols: function block introduced (I), equal to predecessor (=), different behaviour (N), change (C), changes do not effect behaviour (∅), new sensor (NS), complete new behaviour (X).

evolution Sc9, brings workpieces to a ramp. In the previous scenarios, the conveyor belt is just a mechanical ramp, where the workpieces slides down. The pusher components enhance the conveyor and provide sorting functionality during the transportation on the conveyor belt (Figure 4.2b).

Workflow

The normal processing sequence contains following steps for metallic or white workpieces.

1. Stack provides a new workpiece.
2. Crane picks up the workpiece and delivers it to stamp or ramp.
3. Since Sc3, the Stamp imprints and provides the imprinted workpiece for pickup by the crane.
4. Until Sc9, the crane delivers the workpiece from the stamp to a mechanical ramp.
5. Since Sc9, the crane delivers the workpiece to the conveyor, that delivers it to the ramp.
6. Since Sc10, the pusher can sort the workpiece onto different ramps.

The only difference for black workpieces, they are not stamped and get directly delivered to the ramp or conveyor belt. The behaviour becomes more specialized and optimized during the evolution, often by introducing new branches in SFCs. The user interaction is limited to an emergency stop and a start button for each hardware component. The start buttons take place in the initialization of the system. Each function block has a barrier step, in which it keeps active until every function block has reached his barrier step and the operator has pressed all start buttons. This is the only common synchronization point of the system. Every other synchronizations or communications between the instances of the function blocks are done via the physical environment in form of waiting for sensors values and setting actuators variables.

4.1 Scenarios

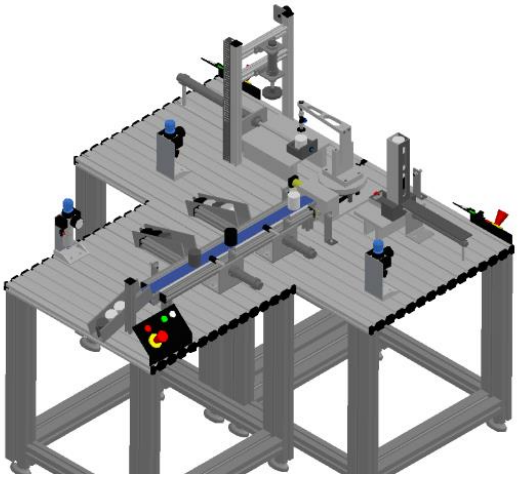
In this section we present the scenarios with software changes. For every scenario we explain the intention of the revision, the differences in the soft- and hardware, along with the invariant formula and his explanation. If the equivalence does not span all output variable, we consider the justification for the assumption of the equivalence on input variables. We shorten the concrete program variables in the formula accordingly to Appendix B. We introduce a short notation for the equivalence of variables between two programs, similar to equivalence of vectors

$$\bigwedge_{0 \leqslant k \leqslant m} v_k^a = v_k^b \quad \equiv \langle v_0, v_1, \ldots, v_m \rangle_{a,b}, \tag{4.1}$$

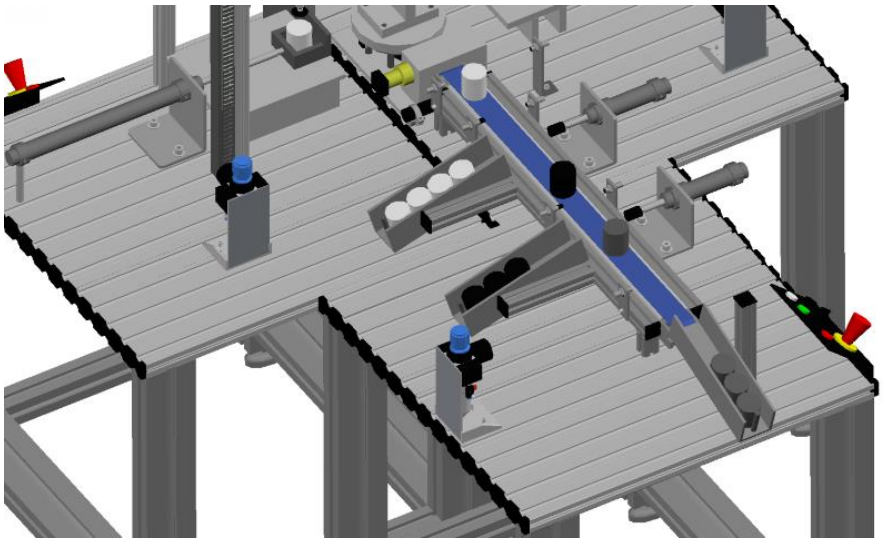
where a_k denotes output variables. For example, $\langle x, y \rangle_{a,b}$ claims the equivalence of the variables x and y in both software revisions a, b

$$x^a = x^b \wedge y^a = y^b. \tag{4.2}$$

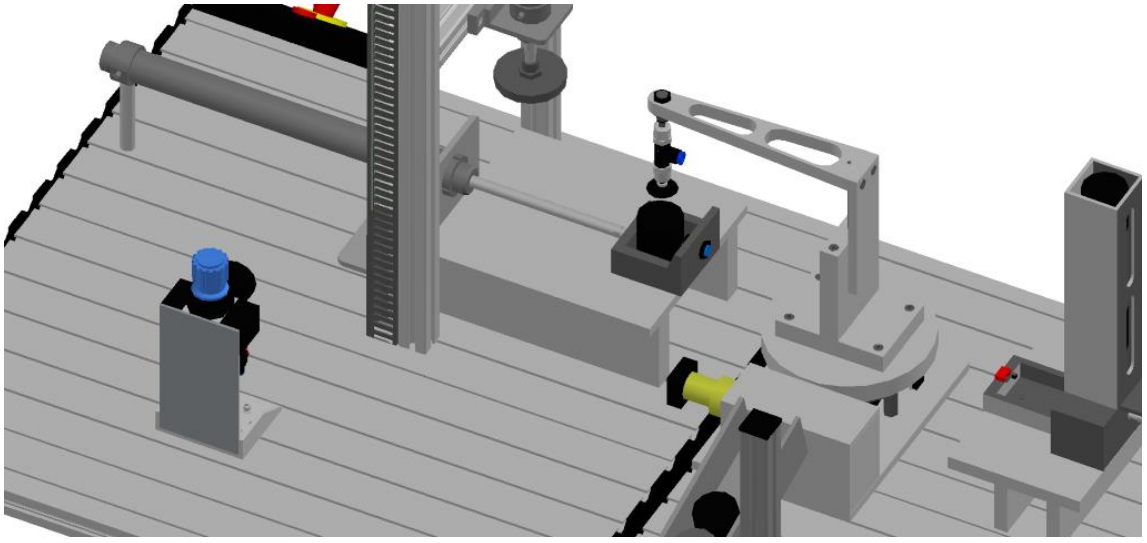
The software revision a, b of the output variable should be clear in the coming contexts, hence we omit the index.



(a) Overview of all hardware components and arrangement of the full PPU (Sc10 or higher)



(b) Conveyor belt with two pushers on the right side. Three ramps store the processed workpieces.



(c) From left to right: Stamp, Crane and Stack. Stamp has a slider for getting and giving workpieces from the crane. The slider from stack provides workpieces from the magazine to the crane. At the bottom edge begins the conveyor belt.

Figure 4.2 Components of the PPU from [VH+14]

Sc0 ≋ Sc1

The change between the Sc0 and Sc1 is an Y-ramp, a mechanical hardware part. The software change corrects a typo in a variable name in the SFC Crane. The variable is intern and not a sensor value, so the effect is limited to the function block. We do not need a condition to prove the equivalence and equivalence spans over every output variable.

$$Inv_{0,1} = \langle cl_A, ctc_A, ctcc_A, mgl_A, ms_A, mvon_A, mvoff_A \rangle \tag{4.3}$$

Sc1 ≋ Sc2

A new capacitive sensor is introduced for detecting metallic workpieces. Preparation for the next evolution is made in the SFC Crane. A new branch, with checks for metallic workpieces, is introduced, but the behaviour is copied from the other branch. We need to remove two unnecessary steps in Sc2 to bypass drifts. With the change both scenarios behaves the same with no equivalence condition and in every output variable. The invariant is the same as in (4.3).

Sc2 ≋ Sc3

The plant starts to imprint every metallic workpiece. The behaviour for plastic workpieces stays the same. A stamp introduces new sensors and actuators variables. The SFC Crane decides with capacity sensor, introduced in Sc2, if the crane jib moves to the stamp. The stamps retrieves the metallic workpieces, imprints a logo and provides it back to the crane. During the stamping the crane waits. Afterwards the crane delivers the workpiece onto the ramp. The scenario effects the behaviour within SFC Crane, the f common synchronization barrier and introduces a new SFC Stamp. We need to add the following conditions to cut out the changes, justified by the requirements description.

- There is no metallic workpiece (4.4) and (4.5).
- The *StartVar* (synchronization variable) are the same in both revisions (4.6).
- If the new emergency stop at the stamp is pressed, an emergency stop must occur in Sc2, too. Be aware of the negative logic of the emergency stops in (4.7).

Additionally, there is a bug fix in Sc3. The bug could cause a moving crane jib, while the hook moves, too. For this fix the developers set the actuators ctc_A and $ctcc_A$ one step earlier in Sc3 than in Sc2. We relax both variables with the except claim. For readable we introduce a $state(\cdot)$ for determining the current step of a SFC. In Section 2.4.2 we decode the current active step into the *_state* variable. The index denotes the software revision.

$$\begin{aligned}
\phi_1 &:= \neg mcs_S && (4.4)\\
&\wedge (state(Crane_2) = state(Crane_3) = \text{Interstep_Check_Workpiece}) \rightarrow wp_s && (4.5)\\
&\wedge \langle Crane.StartVar, Mag.StartVar \rangle && (4.6)\\
&\wedge (\neg ses_S \rightarrow \neg mes_S) && (4.7)\\
\alpha_2 &:= state(Crane_2) = \text{Interstep_Check_Workpiece} && (4.8)\\
\omega_2 &:= state(Crane_2) = \text{Step0} && (4.9)
\end{aligned}$$

$$\begin{aligned}
Inv_{2,3} := \text{UNTIL}_{\phi_1}(&\langle acp_A, cl_A, mgl_A, ms_A, mvoff_A, mvon_A \rangle \\
&\wedge \text{EXCEPT}^{\omega_2}_{\alpha_2}(\langle ctc_A, ctcc_A \rangle))
\end{aligned} \tag{4.10}$$

Every stamp's actuator variable has no corresponding variable in Sc2. We assume the equivalence of the input variables, because the stamp is never used, excluded by the ϕ_1.

Sc3 $\approxeq$ Sc5

Sc5 introduces an optimization for reducing waiting time during stamping. In Sc3 the crane waits at the stamp until the stamp has finished. Sc5 uses the waiting time for delivering plastic workpiece from the magazine to the ramp. This is possible as long as the magazine offers plastic workpieces. The main change is a new branch in the SFC Crane. A minor, but indirectly necessary, change takes place in transition guard in SFC Stamp, that we discuss later. We give multiple invariant for the equivalence. The new behaviour can only appear if plastic and metallic workpieces are available. (4.11) prohibits plastic workpieces and (4.12) metallic workpieces, by forcing the magazine capacitive sensor mcs_S to a certain state in every step. The second term is for the change in the SFC Stamp.

$$\phi^1_{3,5} := mcs_S \wedge (state(Stamp_5) = \text{Step2}) \rightarrow (ssf_S \neq coc_S) \tag{4.11}$$
$$\phi^2_{3,5} := \neg mcs_S \wedge (state(Stamp_5) = \text{Step2}) \rightarrow (ssf_S \neq coc_S) \tag{4.12}$$

We can tighten (4.11) and (4.12). The behaviour only occur, if a metallic workpiece is currently imprinted at the stamp and a plastic workpiece is provided by the magazine. This situation is represented by a transition within the SFC Crane.

$$\phi^3_{3,5} := (state(Crane_5) = \text{Crane\_Go\_Up} \rightarrow mgs_S \wedge \tag{4.13}$$
$$(state(Stamp_5) = \text{Step2} \rightarrow (coc_S \wedge \neg ssf_S) = coc_S) \tag{4.14}$$

A change in the SFC Stamp forces the second term (4.14). In Sc3 we observe the stamp is always empty if the crane leaves the stamp position and is over the conveyor belt. Hence, it is sufficient to test if the crane is over the conveyor to check if the stamp is empty

$$coc_S \rightarrow \neg ssf_S. \tag{4.15}$$

This statement is no longer valid in Sc5 and the developers tighten the corresponding transition guard in SFC Stamp by adding ssf_S. In our over approximation we have decoupled the input variables and the second term rectifies this. We could leave out the second term, if we model the physical environment. The equivalence holds for every $\phi^{1,2,3}_{3,5}$

$$Inv^i_{3,5} := \text{UNTIL}_{\phi^i_{3,5}}(\langle acp_A, cl_A, ctc_A, ctcc_A, ms_A, mvoff_A, \tag{4.16}$$
$$mvon_A, sp_A, ssmi_A, ssmo_A, swl_A \rangle). \tag{4.17}$$

The equivalence holds in very output variable. We removed and merged steps to avoid drifts.

Sc5 ≋ Sc7

In preparation for next evolution a new sensor for dividing black and white workpieces, both non-metallic, is introduced. Some smaller changes in the guards of the SFC Crane need to be considered in the equivalence condition.

We have to enforce certain sensor values on multiple transition to avoid divergence between both SFCs Crane in (4.18), (4.19) and (4.20). With (4.21) we ensure equivalence of the capacitive and optical sensor if the crane is in Crane_Go_Up or Step0 step. Additionally, we have to adapt both scenarios to avoid drifts. Between both scenarios, every output variable is equivalent, except $mvoff_A$, which has a different in initialization value. The assumption of equivalence is not effected, because the counter variable $mvon_A$ is still equivalent. The creators of the case study attest the equivalence on the hardware and the observable behaviour. The initialization value of $mvoff_A$ is alternating within next revisions and is not part of the equivalence.

$$\begin{aligned}
\phi_{5,7} := \quad & (state(Crane_7) = \text{TimeDelay} \rightarrow wp_S) && (4.18)\\
\wedge\ & (state(Crane_7) = \text{Magazin_Stop} \rightarrow wp_S) && (4.19)\\
\wedge\ & (state(Crane_7) = \text{Crane_Go_Up} \rightarrow wp_S) && (4.20)\\
\wedge\ & (state(Crane_7) = \text{Crane_Go_Up} \vee state(Crane_7) = \text{Step0}) && (4.21)\\
& \rightarrow mcs_S = mos_S)\\
Inv_{5,7} := \text{UNTIL}_{\phi_{5,7}}(\langle & acp_A, cl_A, ctc_A, ctcc_A, mgl_A, ms_A, mvon_A, && (4.22)\\
& sgl_A, sp_A, ssmin_A, ssmout_A, swl_A\rangle) && (4.23)
\end{aligned}$$

Sc7 ≋ Sc8

This evolution brings new behaviour for white workpieces. The sensors and most branches were introduced in the previous evolution. In Sc8 the white workpieces becomes an imprint at the stamp, like metallic ones, but we have different pressures. The new branch for white workpieces is a duplicate from the metallic behaviour, only with a different pressure value. Both white and metallic branches received a new step for setting different acp_A. We merge the new additional steps with the corresponding successor step to remove the drift. We can give two restrictions, first we exclude the appearance of white workpieces:

$$\begin{aligned}
\phi^1_{7,8} := & state(Crane_8) = \text{Step0} \rightarrow && (4.24)\\
& (\neg mcs_S \wedge \neg mos_S) \vee (mcs_S \wedge mos_S), && (4.25)
\end{aligned}$$

then we can prove equivalence for every output variable, except $mvoff_A$[2].

$$\begin{aligned}
Inv^1_{7,8} := UNTIL_{\phi^1_{7,9}}(\langle & acp_A, cl_A, ctc_A, ctcc_A, mgl_A, ms_A, && (4.26)\\
& mvon_A, sgl_A, sp_A, ssmin_A, ssmout_A, swl_A\rangle) && (4.27)
\end{aligned}$$

If we allow white workpieces

$$\begin{aligned}
\phi^2_{7,9} := & state(Crane_8) = \text{Step0} \rightarrow && (4.28)\\
& (\neg mcs_A \wedge \neg mos_A) \vee (mcs_A \wedge mos_A) \vee (\neg mcs_A \wedge mos_A), && (4.29)
\end{aligned}$$

[2]See the argumentation of soundness in Sc5 ≋ Sc7.

we need to exclude acp_A from the equivalence.

$$Inv^2_{7,8} := UNTIL_{\phi^2_{7,9}}(\langle cl_A, ctc_A, ctcc_A, mgl_A, ms_A, mvon_A, \tag{4.30}$$
$$sgl_A, sp_A, ssmin_A, ssmout_A, swl_A\rangle) \tag{4.31}$$

acp_A has no influence on the sensor values, hence the equivalence on the input variables in both revision is sound.

Sc8 ≋ Sc9

Before the evolution, the crane delivers processed workpieces onto a ramp. In this evolution a new hardware part, the conveyor belt, is introduced. The crane delivers processed workpieces now on the conveyor belt, which transports the workpieces to the ramp. The only interference between the SFC of the conveyor and the other function blocks is the common synchronization step at the beginning of each SFC. So the new function block can disturb the initialization, and the system stutters forever. After the barrier is passed by all function blocks, the equivalence of all output variable holds. In this equivalence the premise is different to the previous ones. We use the after claim.

$$\text{AFTER}_{\alpha_{8,9}}(\langle cl_A, ctc_A, ctcc_A, mgl_A, ms_A, \tag{4.32}$$
$$mvon_A, sgl_A, sp_A, ssmin_A, ssmout_A, swl_A\rangle)), \tag{4.33}$$

where $\alpha_{8,9}$ describes the passing of the synchronization steps in every common SFCs.

$$\alpha_{8,9} := state(Crane_8) = \text{Interstep\_2} \tag{4.34}$$
$$\wedge\ state(Crane_9) = \text{Interstep\_2} \tag{4.35}$$
$$\wedge\ state(Mag_8) = \text{Green\_Lamp} \tag{4.36}$$
$$\wedge\ state(Mag_9) = \text{Green\_Lamp} \tag{4.37}$$
$$\wedge\ state(Stamp_8) = \text{Green\_Lamp} \tag{4.38}$$
$$\wedge\ state(Stamp_9) = \text{Green\_Lamp} \tag{4.39}$$

$mvoff_A$ is not excluded in the equivalence, because of a different initialization value. The soundness of input equivalence assumption still holds for the same reason as above. New actuator variables $sctc_A, scts_A, sglc_A$ and $swlc_A$, for the controlling the conveyor are excluded, too. But they does not impact the input variables.

Sc9 ≋ Sc10

New behaviour for the conveyor belt and a new SFC Pusher takes place in the evolution. The conveyor belt now runs only if a workpiece needs transportation and stops after a fix amount of time. The previous version keeps the conveyor running.

The pusher takes place within the conveyor belt and consists of two mechanical rods. With each rod a workpiece can be moved off onto a separate ramp. All in all, there are now three ramps, one for each rod and one ramp at the of the belt. New sensors take care if a workpiece appears at the beginning of the conveyor belt and for the position of the pneumatic rods. The pusher's task is to fill all ramps with workpieces without forcing a

sorting, hence just an increased storage for processed workpieces. The SFC Pusher is not part of the common synchronization barrier at the system initialization.

We does not need any adaption and can show the equivalence without any condition. But the equivalence does not cover the two pusher pneumatic rods actuators $py1_A, py2_A$ and for the conveyor belt the $scts_A$ actuator. The justification for input variable equivalence is the missing feedback of these output variables to the input variables. Everything after the delivering of a workpiece to the conveyor belt is hid to the system. *mvoff* is also excluded from the reasons above.

$$Inv_{9,10} := \langle acp_A, cl_A, ctc_A, ctcc_A, mgl_A, ml_A, mvon_A, sctc_A, \tag{4.40}$$
$$sglc_A, swlc_A, sgl_A, sp_A, ssmin_A, ssmout_A, swl_A \rangle \tag{4.41}$$

Sc10 ≋ Sc11

SFC Pusher begins the sorting of workpieces into white, black and metallic with different ramps. The program passes new sensor variables, for detecting color and metallic properties of workpieces on the conveyor to the SFC Pusher. The SFC is completely rewritten, so there is no common behaviour of the pneumatic rods (actuator variables $py1_A$, $py2_A$) Everything else is equivalent without adaptions from our side.

$$Inv_{10,11} := \langle acp_A, cl_A, ctc_A, ctcc_A, mgl_A, mvoff_A, \tag{4.42}$$
$$mvin_A, scts_A, swlc_A, sgl_A, sp_A, ssmin_A, ssmo_A, swl_A \rangle \tag{4.43}$$

The assumption for the equivalence holds, because $py1_A$ and $py2_A$ does not effect any input variables.

Sc11 ≋ Sc12

The SFC Pusher has a complete different behaviour, again. Now he tries to bring a white, black and metallic workpiece in every ramp. Previous behaviour reserves for each workpiece kind an own ramp. Equivalence and condition is the same as in $Sc11 \approxeq Sc12$.

$$Inv_{11,12} := \langle acp_A, cl_A, ctc_A, ctcc_A, mgl_A, mvoff_A, \tag{4.44}$$
$$mvin_A, scts_A, swlc_A, sgl_A, sp_A, ssmin_A, ssmo_A, swl_A \rangle \tag{4.45}$$

Sc12 ≋ Sc13

The evolution brings new an angular sensor for the crane position. Until Sc12, three micro switches on disjoint position determines the position of the crane jib. We need a relation between the three boolean sensor values and the angle position. The relation $R_{12,13}$ describes the dependency between the old input variables coc_S, com_S and cps_S and the new angular input acp_S.

```
1  DEFINE
2    Sensor_CranePositionStamp := Sensor_AnalogCranePosition > 0ud16_8160
3                                 & Sensor_AnalogCranePosition < 0
                                      ud16_8260;
4    Sensor_CraneOnConveyor := Sensor_AnalogCranePosition > 0ud16_16160
5                              & Sensor_AnalogCranePosition < 0
                                      ud16_16260;
6    Sensor_CraneOnMagazin := Sensor_AnalogCranePosition > 0ud16_24290
7                             & Sensor_AnalogCranePosition < 0
                                      ud16_24390;
```

Figure 4.3 SMV code for emulate the three boolean sensor variables from the new angular position.

$$R_{12,13} \subseteq \mathbb{B}^3 \times [\alpha, \beta] \tag{4.46}$$

$$(coc_S, com_S, cps_S) R_{12,13} acp_S =_{\text{def}} \begin{cases} acp_S \in [\alpha,\beta] \backslash [a,b] \backslash [c,d] \backslash [e,f]: & \neg coc_S \wedge \neg com_S \wedge \neg cps_S \\ acp_S \in [a,b]: & coc_s \\ acp_S \in [c,d]: & com_S \\ acp_S \in [e,f]: & cps_S \\ acp_S = \boxdot: & (coc_S \wedge com_S) \\ & \vee (com_S \wedge cps_S) \\ & \vee (coc_S \wedge cps_S) \end{cases} \tag{4.47}$$

where coc_S is the position over the conveyor, resp. com_S for magazine and cps_S for stamp. acp_S denotes the angle in some interval $[\alpha, \beta]$. If one of the switches is true, the acp_S has to be in the corresponding interval, non-deterministically. If no switch is triggered, the crane is at a arbitrary position, except in the switch intervals. If two of the boolean micro switches are true, acp_S is undefined. There multiple possibilities for modelling $R_{12,13}$ into SMV. First, we have the three boolean sensor values com_S, coc_S, cps_S and the angular position acp_S as input variables in SMV and exclude every ω-structure, that violates the relation $R_{12,13}$.

$$\phi_{12,13} := (coc_s \rightarrow a \leqslant acp_S \leqslant b \wedge \neg com_S \wedge \neg cps_S) \tag{4.48}$$

$$\wedge\, (com_s \rightarrow c \leqslant acp_S \leqslant d \wedge \neg coc_S \wedge \neg cps_S) \tag{4.49}$$

$$\wedge\, (cps_s \rightarrow e \leqslant acp_S \leqslant f \wedge \neg com_S \wedge \neg coc_S) \tag{4.50}$$

$$Inv_{12,13} := \text{UNTIL}_{\phi_{12,13}}(\langle acp_A, cl_A, ctc_A, ctcc_A, mgl_A, mvoff_A, mvin_A, scts_A, \tag{4.51}$$

$$swlc_A, sgl_A, sp_A, ssmin_A, ssmo_A, swl_A, py1_A, py2_A \rangle). \tag{4.52}$$

Second, the relation $R_{12,13}$ is a function in the direction from right to left, also from acp_S to the three boolean variables. We can use SMV's defines to model this. Figure 4.3 shows the SMV code with the concrete intervals taken from Sc13.

The equivalence between Sc12 and Sc13 covers every actuator value. Additionally we encounter a typo in the interval borders within the SFC Crane, that we fixed for the proof.

4.2 Results

We discuss the model sizes and runtimes from the nuXmv [Cav+14] in version 1.0. The case names consists from the both compared revisions, e.g. "03.05" is the comparison of the

Scenario	#INT	#USINT	#BOOL	Enum [bit]	Total [bit]
0	1	2	60	8	100
1	1	2	60	8	100
2	1	2	70	9	111
3	5	6	116	13	257
5	5	6	117	14	259
7	5	6	117	14	259
8	6	6	117	14	275
9	7	8	140	17	333
10	12	16	173	23	516
11	12	16	185	23	528
12	28	26	199	24	879
13	32	26	197	24	941

Table 4.2 Number of variables in each scenario. The column *Enum* describes the number of bits consumed by the enumeration type variables, correlating with number of SFC states. Column *Total* is the estimation of used bits by the program, where INT takes 16 bits, USINT 8 bits, BOOL 1 bit.

Sc03 against Sc05. An additional tag gives further information about the case. "TA" stands for the application of timer abstraction. We chose first variant of timer abstraction with the guaranteed end of the waiting time (Figure 3.8b). "PM" signals the use of a appropriate logical model of physical environment, that we do not discuss further in this thesis. We have different conditions and proves for the equivalence between "03.05". "03.05+OM" considers only metallic workpieces, "03.05+OP" only plastic workpiece and "03.05+PMr" uses a physical model, so we do not need Equation (4.14). The results are in Tables 4.2 and 4.3.

Spaces and Variables

The case study introduces with every evolution step additional hardware or complexity to the behaviour. This lead to an increasing state and input space. Figure 4.5 shows the growth of bits in the model checker. Another influence of the growth is the promotion every integer variable to a global fixed bit width, due to the typing in nuXmv. SMV is very strict in data types, in despite to IEC 61131-3, the promotion makes the various expression compatible. Of course, this solution is not applicable if the origin software relies on overflows. We needed a higher bit width in the later scenarios, to due a higher fixed bit width. For this reason we give in Figure 4.4 and Table 4.2 a view on the declared variables. The "TA" implementation reduces some bits, because the internal variables storing the waiting time are not needed.

Performance

We measured the CPU time consumed by nuXmv to prove the equivalence between the scenarios. The used computer has a Intel® Core™ i7-860 with 2.80GHz on four physical

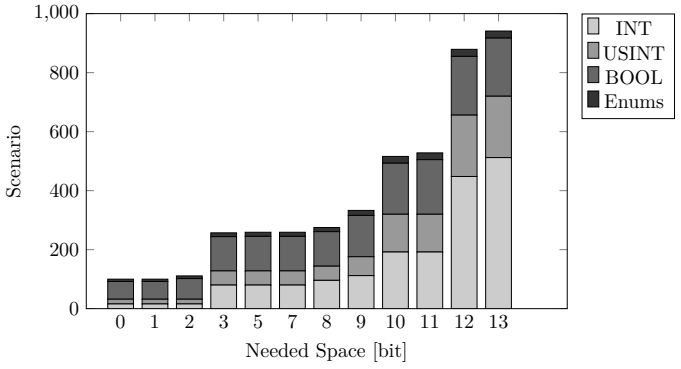


Figure 4.4 The visualization of Table 4.2 in number of bits: integer variables ■, unsigned small integers ■, boolean ■ and enumerations ■.

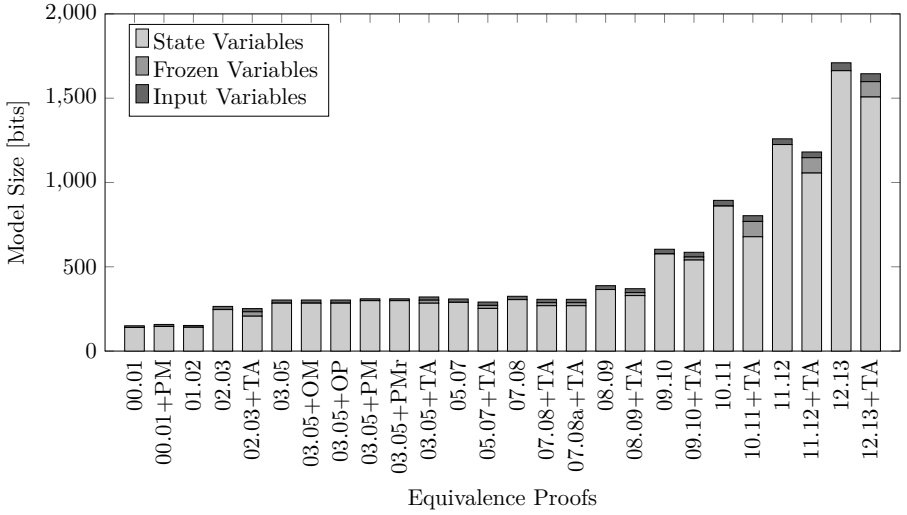


Figure 4.5 Number of bits for state □, frozen ■ and input ■ variables in the model checker nuXmv. This correlates with size of the declared variables in Figure 4.4 and the running times of the model checker in Figure 4.6.

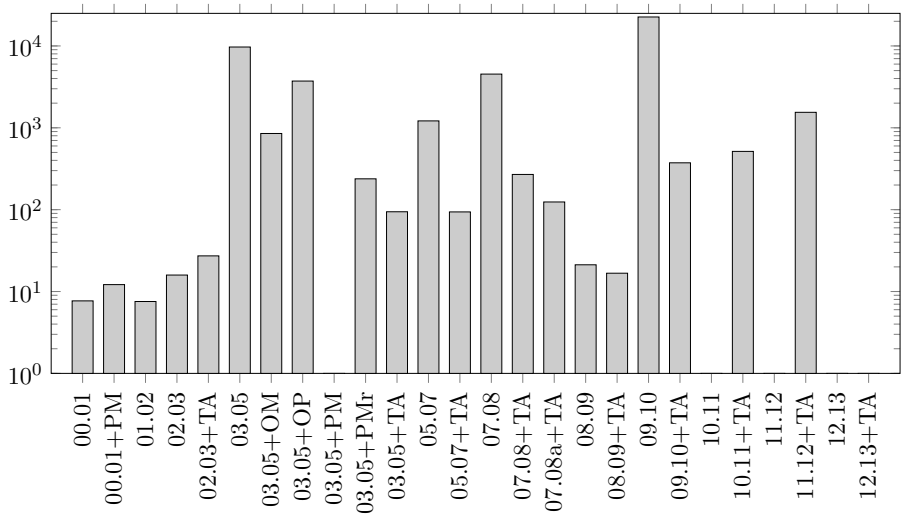


Figure 4.6 Diagram of the CPU time from Table 4.3. Please note that the y axis has a logarithmic scale.

cores and 8 gigabytes of memory. For the benchmark we set a timeout of 10 hours and disable the address randomization. The address randomization causes non-deterministic performances of nuXmv on our systems. Sometimes a run, that completed in half hour, needed several hours to complete. Disabling address randomization gives reproducible performance, but the runtimes in Figure 4.6 are not the best run we had. Some proves did not finish under the 10 hours timeout. Additionally to the mean μ and standard derivation σ inTable 4.3 we give minimum and maximum runtimes with address randomization from a different system, Intel® Dual Core E5400 with 2.70 GHz and 4 GB memory. These results show the range of the runtimes in practise and how runtimes can diverge between the same model (Figure 4.7).

	Used Bits				CPU Time			
Case	Frozen	Input	State	Total	μ	σ	min	max
00.01	0	10	140	150	0m 7s	0m 0s	0m 4s	0m 7s
00.01+PM	0	12	146	158	0m 12s	0m 0s	0m 6s	0m 12s
01.02	0	11	141	152	0m 7s	0m 0s	0m 4s	0m 7s
02.03	0	19	246	265	0m 15s	0m 0s	0m 9s	0m 17s
02.03+TA	26	19	207	252	0m 27s	0m 0s	0m 15s	0m 28s
03.05	0	19	284	303	161m 42s	0m 29s	15m 7s	9325m 44s
03.05+OM	0	19	284	303	14m 12s	0m 15s	8m 55s	544m 30s
03.05+OP	0	19	284	303	62m 4s	0m 29s	18m 8s	779m 3s
03.05+PM	0	11	299	310	–	–	25m 41s	6246m 48s
03.05+PMr	0	11	299	310	3m 58s	0m 0s	1m 57s	21m 2s
03.05+TA	18	19	284	321	1m 34s	0m 0s	0m 59s	1m 42s
05.07	0	20	289	309	20m 14s	0m 27s	13m 42s	20m 51s
05.07+TA	18	20	253	291	1m 33s	0m 0s	0m 45s	1m 39s
07.08	0	20	305	325	75m 29s	0m 26s	50m 31s	78m 56s
07.08+TA	18	20	269	307	4m 29s	0m 6s	2m 51s	5m 5s
07.08a+TA	18	20	269	307	2m 4s	0m 3s	1m 19s	2m 19s
08.09	0	23	365	388	0m 21s	0m 0s	0m 13s	0m 24s
08.09+TA	18	23	329	370	0m 16s	0m 0s	0m 10s	0m 19s
09.10	0	28	576	604	376m 24s	2m 24s	212m 17s	379m 9s
09.10+TA	18	28	540	586	6m 13s	0m 0s	3m 38s	6m 54s
10.11	0	34	860	894	–	–	1330m 8s	3382m 13s
10.11+TA	91	34	678	803	8m 35s	0m 9s	4m 26s	9m 47s
11.12	0	34	1225	1259	–	–	1311m 59s	1311m 59s
11.12+TA	91	34	1056	1181	25m 45s	0m 6s	11m 40s	27m 42s
12.13	0	47	1663	1710	–	–	1327m 26s	1327m 26s
12.13+TA	91	47	1507	1645	–	–	15m 40s	33m 51s

Table 4.3 Equivalence with the number of needed bits (frozen, input and state variables) and the consumed cpu time for proofing the equivalence with nuXmv and IC3.

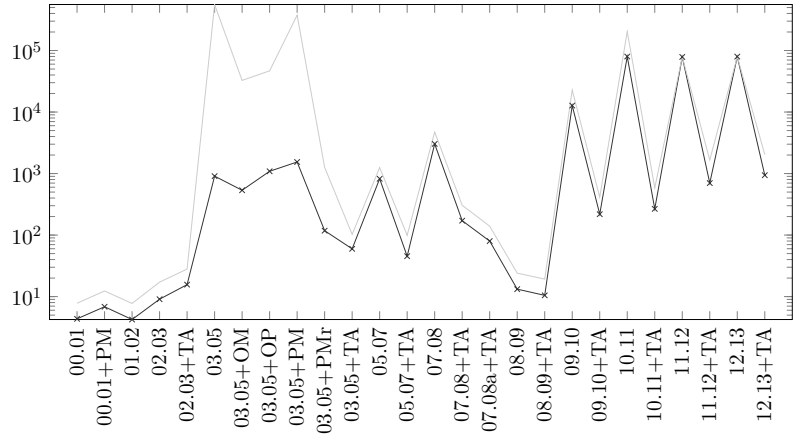


Figure 4.7 Minimum and maximum runtime with address randomization from Table 4.3. Please note that the y axis has a logarithmic scale.

5. Conclusion

Summary

This is the first work on regression verification on automation system. We presented formalization and operational semantics for a subset of Structured Text (ST) and Sequential Function Chart (SFC). The subset is determined on the used IEC 61131 elements in the case study [VH+14]. SFC with simultaneously divergence and convergence (fork and join) are reducible to multiple SFC without the use of fork and joins, we reached SFC_0. We translated ST and SFC_0 into an intermediate representation ST_0, with a limitation to assignments and if statement as long with boolean and integer variables. ST_0 does not contain loops or function block calls, just one large program body. Especially, the translation is possible if every loop in ST code or SFC actions are unwindable. ST_0 is our intermediate language and the base for the translation into SMV. The equivalent big ST_0 programs were translated into separate SMV modules, which these modules were bisimulated. We described the equivalence obligation as invariants for solving with IC3 technique. The advantage of IC3 is his performance in relation to explicit state model checker. IC3 just tries to find a inductive clauses, that supports the safety property and forces us to write the equivalence obligation as an invariant, hence we are not allowed to verify arbitrary LTL formulae.

We introduced four claims, UNTIL, AFTER, WITHIN and EXCEPT, with corresponding SMV code for modelling the conditional equivalences in the case study. The case study [VH+14] was our playground for the developing and testing of general techniques for the regression verification in the PLC context. We proved the equivalence between eleven evolution steps of the case study. Every proof has his own condition for the restriction of input and states values. The complexity of the condition is a factor in the performance of the model checker. The results shows acceptable runtime durations and potentials of timer abstractions. In the theoretical section we discuss the embedding the PLC within a cyber-physical system. This leaded to different perspectives on equivalence.

Limitation

Our developed techniques are suitable for the case study. They should be applicable for a width spectrum of automation code, or at least extensible for support more language features. We didn't solve the drifts satisfactorily. Our solution is the removing of the delayed steps, after consulting the PPU authors. The delayed step are normally not desired and considered as bad programming habit. With a scan cycle of four milliseconds the drifts are not observable by humans, but in the detail there is a different triggering of the actuators. Our techniques are applicable if the changes are relative small. Bigger changes lead to more restrict conditions and the equivalence loose significance. For example we failed in the equivalence of SFC Pusher in the case study. His behaviour is not describable

on his both output variables, we need to talk about quantity and kind of workpieces at the ramps.

Future Work

Drifts are an open question. They lead to a fuzzyfication of the time, that means the comparison between output variables it not limited within one turn. Instead the comparison talks about output variables from previous and next turns with increasing distance in time. Independent of the used approach for solving drifts, the discussion of the approach's soundness will be central point in his justification.

We can try to find a common behaviour on a higher level, such as the external observable state of the plant. This is always possible as an abstraction, but hard to realize and the equivalence formulation can be weak. For example, the SFC Pusher within the case study was not part of the equivalence. Every revision changed the sorting order of the workpiece, but all revisions try to fill the ramps evenly in best effort.. In general the equivalence on the observable behaviour seems more comprehensible and understandable for engineers.

Our presented equivalence conditions were not understandable from the case study's authors. There is a gap between the formal verification on the software level and the perspective of the requirement changes. One way is the computation the weakest condition for an equivalence and the check if this condition meets the engineer expectations. We showed a simple syntactical approach for finding a equivalence condition between two SFC. Furthermore, the model checker returns counterexamples if the equivalence is not valid. The counterexamples contains information of the critical and unequal paths. We excluded these paths by given manual derived conditions. However, IC3 uses the counterexamples to strengthen his induction and we could infer from the induction an equivalence condition.

Another way will be to give the engineer high level syntactical tools for the comparison to describe the equivalence between to scenario, maybe in a guided manner. Our intention with four equivalence claims goes in this direction of bringing the verification nearer to the software or requirements level of the developing process. One big topic will be the useability of verification methods for the engineers.

Nevertheless this thesis shows, that the regression verification of PLC is feasible, and gives a foundation for future work in this area.

A. Introductory Example

A.1 Software

```
1  PROGRAM main
2     VAR cbc : ConveyorBeltExample_I;
3         EmergencyStop    : bool;
4         Sensor_w1        : bool;
5         Sensor_w2        : bool;
6         Actuator_run     : bool;
7         Actuator_pickup  : bool;
8     END_VAR
9     (* passing sensors to SFC *)
10    cbc.w1 := Sensor_w1;
11    cbc.w2 := Sensor_w2;
12
13    IF  EmergencyStop
14    THEN       (* panic! resetting the system *)
15      Actuator_pickup := FALSE;
16      Actuator_run    := FALSE;
17      cbc.SFCReset := TRUE;
18    ELSE       (* setting actuators from SFC *)
19      cbc();
20      Actuator_Pickup := cbc.pickup;
21      Actuator_Conveyor  := cbc.run;
22    END_IF
23 END_PROGRAM
```

Figure A.1 Example program for handling emergency stops, sensor and actuator values for the SFC in Figure 2.17a and Figure A.2

```
1  TYPE      ConveyorBeltExample_I_TYPE : {Wait, Run, Pick}; END_TYPE
2  FUNCTION_BLOCK ConveyorBeltExample_I
3    VAR _state    : ConveyorBeltExample_I_TYPE;
4        _transit  : bool;
5        run : bool;
6        pickup : bool;
7    END_VAR
8
9    CASE _state OF
10   Wait:
11     IF _transit THEN (* empty *) END_IF;
12     _transit := FALSE;
13      run = FALSE;
```

```
14       IF w1 THEN
15         _transit := TRUE;
16         _state := Run
17       END_IF
18
19     Run:
20       IF _transit THEN (* empty *) END_IF;
21       _transit := FALSE;
22       run = true;
23       IF w2 THEN
24         _transit := TRUE;
25         _state := Pick
26       END_IF
27
28     Pick:
29       IF _transit THEN (* empty *) END_IF;
30       _transit := FALSE;
31       run = FALSE;
32       pickup = TRUE;
33       IF NOT w2  THEN
34         _transit := TRUE;
35         _state := Wait
36       END_IF
37     END_IF
38   END_FUNCTION_BLOCK
```

Figure A.2 Structured Text for the SFC in figure 2.17a

```
1   PROGRAM main
2     VAR  EmergencyStop    : bool;
3          Sensor_w1        : bool;
4          Sensor_w2        : bool;
5          Actuator_run     : bool;
6          Actuator_pickup  : bool;
7          cbc$_state    : ConveyorBeltExample_I_TYPE;
8          cbc$_transit  : bool;
9          cbc$run : bool;
10         cbc$pickup : bool;
11    END_VAR
12    cbc$w1 := Sensor_w1;
13    cbc$w2 := Sensor_w2;
14
15    IF  EmergencyStop
16    THEN       (* panic! resetting the system *)
17      Actuator_pickup := FALSE;
18      Actuator_run    := FALSE;
19      cbc$SFCReset := TRUE;
20    ELSE       (* setting actuators from SFC *)
21      IF cbc$_state = Wait  THEN
22        cbc$run = FALSE;
23        cbc$pickup = FALSE;
24        IF cbc$w1 THEN
25          cbc$_state := Run
26        END_IF
27      ELSEIF cbc$_state = Run THEN
28        cbc$run = TRUE;
29        IF cbc$w2 THEN
30          cbc$_state := Pick
31        END_IF
32      ELSEIF cbc$_state = Pick THEN
33        cbc$run = FALSE;
34        cbc$pickup = TRUE;
35        IF NOT cbc$w2  THEN
36          cbc$_state := Wait
```

```
37        END_IF
38      END_IF
39
40      Actuator_Pickup := cbc$pickup;
41      Actuator_Conveyor  := cbc$run;
42    END_IF
43  END_PROGRAM
```

Figure A.3 The complete introductory example in ST_0

A.2 SMV

```
1  MODULE main
2
3  IVAR
4      w1 : boolean;  w2 : boolean; Bad : boolean;
5
6  VAR
7      v1 : V1(w1,w2);
8      v2 : V2(w1,w2,Bad);
9      premise : boolean;
10
11 ASSIGN
12     init(premise) := TRUE;
13     next(premise) := premise & {{phi}} ;
14
15 INVARSPEC  premise -> v1.pickup = v2.pickup & v1.run = v2.run;
16
17 LTLSPEC G (  premise -> v1.pickup = v2.pickup & v1.run = v2.run );
18
19 MODULE V1(w1, w2)
20 VAR
21     _state : {Wait, Run, Pick};
22     run : boolean;
23     pickup : boolean;
24
25 ASSIGN
26     init(_state) := Wait;
27     next(_state) := case
28                         _state = Wait & w1   : Run;
29                         _state = Run  & w2   : Pick;
30                         _state = Pick & ! w2 : Wait;
31                         TRUE : _state;
32                     esac;
33
34     init(run) := FALSE;
35     next(run) := case
36                         _state = Run : TRUE;
37                         TRUE : FALSE;
38                 esac;
39
40     init(pickup) := FALSE;
41     next(pickup) := case
42                         _state = Pick : TRUE;
43                         TRUE : FALSE;
44                     esac;
45
46 MODULE V2(w1, w2, Bad)
47 VAR
48     _state : {Wait, Run, Pick, Reject};
49     b : boolean;
50     run : boolean;
51     pickup : boolean;
```

```
52
53  ASSIGN
54      init(_state) := Wait;
55      next(_state) := case
56                          _state = Wait &  w1      : Run;
57                          _state = Run  &  w2 & !b : Pick;
58                          _state = Run  &  w2 &  b : Reject;
59                          _state = Pick & !w2      : Wait;
60                          _state = Reject & !w2    : Wait;
61                          TRUE : _state;
62                      esac;
63
64      init(run) := FALSE;
65      next(run) := case
66                       _state = Run | _state = Reject : TRUE;
67                       TRUE : FALSE;
68                  esac;
69
70      init(pickup) := FALSE;
71      next(pickup) := case
72                          _state = Pick : TRUE;
73                          TRUE : FALSE;
74                      esac;
75
76      init(b) := FALSE;
77      next(b) := case
78                     _state = Run : b | Bad;
79                     _state = Wait : FALSE;
80                     TRUE : b;
81                 esac;
```

Figure A.4 SMV modules for the introductory example in Section 1.1.

B. Abbreviation for Variables

Complete Name	*Abbreviation*
Actuator_AnalogCranePressure	acp_A
Actuator_CraneLower	al_A
Actuator_CraneTurnClockwise	ctc_A
Actuator_CraneTurnCounterclockwise	$ctcc_A$
Actuator_MagazinGreenLamp	mgl_A
Actuator_MagazinSlider	ms_A
Actuator_MagazinVacuumOff	$mvoff_A$
Actuator_MagazinVacuumOn	$mvon_A$
Actuator_PusherY1	$py1_A$
Actuator_PusherY2	$py2_A$
Actuator_SorterConveyorTowardsCrane	$sctc_A$
Actuator_SorterConveyorTowardsStacker	$scts_A$
Actuator_SorterGreenLampConveyor	$sglc_A$
Actuator_SorterWhiteLampConveyor	$swlc_A$
Actuator_StampGreenLamp	sgl_A
Actuator_StampPusher	sp_A
Actuator_StampSliderMovedIn	$ssmi_A$
Actuator_StampSliderMovedOut	$ssmo_A$
Actuator_StampWhiteLamp	swl_A
Sensor_AnalogCranePosition	acp_S
Sensor_CraneDown	cd_S
Sensor_CraneOnConveyor	coc_S
Sensor_CraneOnMagazin	com_S
Sensor_CranePositionStamp	cps_S
Sensor_CraneSucked	cs_S
Sensor_CraneUp	cu_S
Sensor_MagazinCapacitiveSensor	mcs_S
Sensor_MagazinEmergencyStop	mes_S
Sensor_MagazinOpticalSensor	mos_S
Sensor_SliderMovedOut	smo_S
Sensor_SliderNotMovedOut	$snmo_S$
Sensor_SorterCapacitiveSensorPusher1	$scsp1_S$
Sensor_SorterCapacitiveSensorPusher2	$scsp2_S$

Sensor_SorterEmergencyStop	ses_S
Sensor_SorterLightbarrierCraneInterface	$slci_S$
Sensor_SorterLightnessSensorPusher1	$slsp1_S$
Sensor_SorterLightnessSensorPusher2	$slsp2_S$
Sensor_SorterLightnessSensorPusher3	$slsp3_S$
Sensor_SorterLightnesssensorCraneInterfaceInverse	$slcii_S$
Sensor_SorterPusher1MovedIn	$sp1mi_S$
Sensor_SorterPusher1MovedOut	$sp1mo_S$
Sensor_SorterPusher2MovedIn	$sp2mi_S$
Sensor_SorterPusher2MovedOut	$sp2mo_S$
Sensor_SorterStartButton	ssb_S
Sensor_SorterSwitchManuellAutomatic	$ssma_S$
Sensor_StampEmergencyStop	ses_S
Sensor_StampLowered	sl_S
Sensor_StampSliderFilled	ssf_S
Sensor_StampSliderSensorMovedIn	$sssmi_S$
Sensor_StampSliderSensorMovedOut	$sssmo_S$
Sensor_StampStartButton	ssb_S
Sensor_StampUp	su_S
Sensor_StartButtonMagazin	sbm_S
Sensor_WorkpieceReady	wp_S

Table B.1 Abbreviation for sensor and actuator variables for the equivalence conditions of the PPU case study

Bibliography

[BK08] Christel Baier and Joost-Pieter Katoen. “Principles of Model Checking”. In: MIT press Cambridge, 2008. Chap. Linear Temporal Logic, pp. 229–312.

[Bau+04a] Nanette Bauer, Ralf Huuck, Ben Lukoschus, and Sebastian Engell. “A Unifying Semantics for Sequential Function Charts”. English. In: *Integration of Software Specification Techniques for Applications in Engineering*. Ed. by Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper. Vol. 3147. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 400–418. ISBN: 978-3-540-23135-6. DOI: 10.1007/978-3-540-27863-4_22. URL: http://dx.doi.org/10.1007/978-3-540-27863-4_22.

[Bau+04b] Nanette Bauer, Sebastian Engell, Ralf Huuck, Sven Lohmann, Ben Lukoschus, Manuel Remelhe, and Olaf Stursberg. “Verification of PLC Programs Given as Sequential Function Charts”. English. In: *Integration of Software Specification Techniques for Applications in Engineering*. Ed. by Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper. Vol. 3147. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 517–540. ISBN: 978-3-540-23135-6. DOI: 10.1007/978-3-540-27863-4_28. URL: http://dx.doi.org/10.1007/978-3-540-27863-4_28.

[Bor+00] Sébastien Bornot, Ralf Huuck, and Ben Lukoschus. “Verification of Sequential Function Charts Using SMV”. In: *PDPTA*. Ed. by Hamid R. Arabnia. CSREA Press, 2000.

[Bra11] Aaron R. Bradley. “SAT-Based Model Checking without Unrolling”. English. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Vol. 6538. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 70–87. ISBN: 978-3-642-18274-7. DOI: 10.1007/978-3-642-18275-4_7. URL: http://dx.doi.org/10.1007/978-3-642-18275-4_7.

[Bri+02] Ed Brinksma, Angelika Mader, and Ansgar Fehnker. “Verification and Optimization of a PLC Control Schedule”. English. In: *International Journal on Software Tools for Technology Transfer* 4.1 (2002), pp. 21–33. ISSN: 1433-2779. DOI: 10.1007/s10009-002-0079-0. URL: http://dx.doi.org/10.1007/s10009-002-0079-0.

[Bur+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. “Symbolic model checking: 10^{20} States and beyond”. In: *Information and Computation* 98.2 (1992), pp. 142 –170. ISSN: 0890-5401. DOI: http://dx.doi.org/10.1016/0890-5401(92)90017-A. URL: http://www.sciencedirect.com/science/article/pii/089054019290017A.

[Cav+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. "The nuXmv Symbolic Model Checker". In: *CAV*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 334–342. ISBN: 978-3-319-08866-2.

[Com02a] International Electrotechnical Commission. *IEC 60848: GRAFCET specification language for sequential function charts*. Tech. rep. International Electrotechnical Commission, 2002.

[Com02b] International Electrotechnical Commission. *IEC 61131: Programmable controllers – Part 3: Programming languages*. Tech. rep. International Electrotechnical Commission, Feb. 2002.

[Cyt+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "An Efficient Method of Computing Static Single Assignment Form". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, pp. 25–35. ISBN: 0-89791-294-2. DOI: 10.1145/75277.75280. URL: http://doi.acm.org/10.1145/75277.75280.

[Fel+14] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. "Automating Regression Verification". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. ACM, 2014, pp. 349–360.

[GS08] Benny Godlin and Ofer Strichman. "Inference rules for proving the equivalence of recursive procedures". English. In: *Acta Informatica* 45.6 (2008), pp. 403–439. ISSN: 0001-5903. DOI: 10.1007/s00236-008-0075-2. URL: http://dx.doi.org/10.1007/s00236-008-0075-2.

[GS09] Benny Godlin and Ofer Strichman. "Regression Verification". In: *Proceedings of the 46th Annual Design Automation Conference*. DAC '09. San Francisco, California: ACM, 2009, pp. 466–471. ISBN: 978-1-60558-497-3. DOI: 10.1145/1629911.1630034. URL: http://doi.acm.org/10.1145/1629911.1630034.

[GS13] Benny Godlin and Ofer Strichman. "Regression Verification: Proving the Equivalence of similar Programs". In: *Software Testing, Verification and Reliability* 23.3 (2013), pp. 241–258. ISSN: 1099-1689. DOI: 10.1002/stvr.1472. URL: http://dx.doi.org/10.1002/stvr.1472.

[Hol97] Gerard J Holzmann. "The model checker SPIN". In: *IEEE Transactions on software engineering* 23.5 (1997), pp. 279–295.

[LC+99] S. Lampérière-Couffin, O. Rossi, J.-M. Roussel, and J.-J. Lesage. "Formal Validation of PLC programs: a survey". In: *European Control Conference 1999, ECC'99,* Karlsruhe (Germany), 1999, CD–ROM paper n°741.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 032114306X.

[McM03] K.L. McMillan. "Interpolation and SAT-Based Model Checking". English. In: *Computer Aided Verification*. Ed. by Jr. Hunt WarrenA. and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 1–13. ISBN: 978-3-540-40524-5. DOI: 10.1007/978-3-540-45069-6_1. URL: http://dx.doi.org/10.1007/978-3-540-45069-6_1.

[Nar+10a] Flor Narciso, Addison Rios-Bolivar, Francisco Hidrobo, and Olga Gonzalez. "A syntactic specification for the programming languages of the IEC 61131-3 standard". In: *Proceedings of the 9th WSEAS international conference on computational intelligence, man-machine systems and cybernetics*. World Scientific, Engineering Academy, and Society (WSEAS). 2010, pp. 171–176.

[Nar+10b] Flor Narciso, Addison Rios-Bolivar, Francisco Hidrobo, and Olga Gonzalez. "A syntactic specification for the programming languages of the IEC 61131-3 standard". In: *Proceedings of the 9th WSEAS international conference on computational intelligence, man-machine systems and cybernetics*. World Scientific, Engineering Academy, and Society (WSEAS). 2010, pp. 171–176.

[Neu+00] Peter Neumann, Eberhard E. Grötsch, Christoph Lubkoll, and Rene Simon. "SPS-Standard: IEC61131". German. In: vol. 3. Oldenbourg, 2000. Chap. 4 Programmiersprachen, pp. 125–264.

[Sme+00] O. De Smet, S. Couffin, O. Rossi, G. Canet, J.-J. Lesage, P. Schnoebelen, and H. Papini. "Safe programming of PLC using formal verification methods". In: *4th International PLCopen conference on Industrial Control Programming, ICP'2000*. Utrecht (The Netherlands), 2000, pp. 73–78.

[SB11] Fabio Somenzi and Aaron R Bradley. "IC3: Where Monolithic and Incremental Meet". In: *FMCAD*. 2011, pp. 3–8.

[Str09] Ofer Strichman. "Regression Verification: Proving the Equivalence of Similar Programs". English. In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 63–63. ISBN: 978-3-642-02657-7. DOI: 10.1007/978-3-642-02658-4_8. URL: http://dx.doi.org/10.1007/978-3-642-02658-4_8.

[TJ09] Michael Tiegelkamp and Karl Heinz John. *SPS-Programmierung mit IEC 61131-3*. German. 4., neubearbeitete Auflage. VDI-Buch. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-00269-4.

[VH+14] Birgit Vogel-Heuser, Christoph Legat, Jens Folmer, and Stefan Feldmann. *Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit*. Tech. rep. Institute of Automation and Information Systems, Technische Universität München, 2014. URL: https://mediatum.ub.tum.de/node?id=1208973.

[Wik13] Wikipedia. *Default Logic*. [Online; accessed 05-Nov-2014]. 2013. URL: http://en.wikipedia.org/wiki/Default_logic.

[YF03] M Bani Younis and Georg Frey. "Formalization of existing PLC programs: A survey". In: *Proceedings of CESA*. 2003, pp. 0234–0239.

List of Figures

List of Definitions and Theorems